IAN SINCLAIR

THE ZX SPECTRUM

HOW TO USE AND PROGRAM

The ZX Spectrum
How to Use and Program

IAN SINCLAIR

The ZX Spectrum How to Use and Program

PANTHER
Granada Publishing

Panther Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

Published by Panther Books 1983

A Granada Paperback Original

British Library Cataloguing in Publication Data
Sinclair, Ian
The ZX Spectrum: how to use and program
1. Sinclair ZX Spectrum (Computer)
I. Title
001.64'04 QA76.8.S625

ISBN 0-586-06104-5

Printed and bound in Great Britain by
Cox & Wyman Ltd, Reading

Contents

Preface

This book has been planned as an introduction to Spectrum computing for the family user. No book of less than encyclopaedia length can hope to deal with all the actions of a modern microcomputer in detail, and this is no exception. What I have done is to concentrate on the features of the Spectrum which are of most interest and utility to the family user, as distinct from the educational, scientific or engineering user. These selected features are dealt with in detail, using examples which are short and easy to type, and they are recalled in the games program of Chapter 10 and the information storage program of Chapter 11.

From this launch-pad, you can go where you please! When you have completed this book, you will know what Spectrum can do and you will be better able to decide what you want of your computer. You will also find that you have gained a better understanding of the Manual for the Spectrum so that you can, if you wish, delve into the instructions that are intended to be used by the more experienced programmer.

Ian Sinclair

Chapter 1
Hardware

The hardware of computing consists of all the bits that you can drop and spill coffee over. For the Spectrum, that means the computer itself, the ZX power supply, and the connecting leads. The first action that you have to see to is connecting a plug to the mains cable of the ZX power supply. This, for most houses in Britain, means connecting a three-pin mains plug of the type that is shown in Fig. 1.1. There are

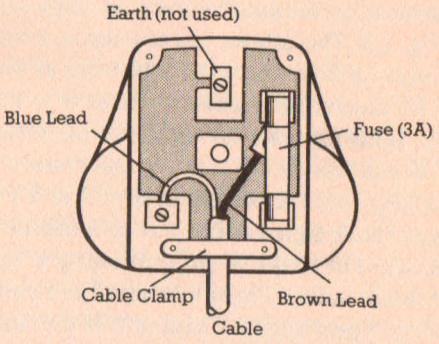


Fig. 1.1. Connecting the mains plug.

only two connections to make, the live and the neutral. The live lead is colour-coded brown (a very light brown) and the neutral lead is coded blue. They should be connected as indicated in Fig. 1.1. If you haven't connected up a three-pin plug before, or if you have any doubts, take the plug and the ZX power supply to an electrician to have the plug fitted.

With that hurdle over, you are almost ready to work some Spectrum magic, but you need the use of a TV receiver. A computer is a device which is arranged so as to send signals to a TV receiver, and unless you connect a TV receiver to the Spectrum you won't be able to

see what the Spectrum is doing. It will still compute for you just as well, but you won't see what is going on.

The Spectrum comes with its TV cable ready to attach, with an aerial plug at one end of the lead, and a different type of plug (a phono plug) at the other end. The two different plugs are illustrated in Fig. 1.2. You could, of course, simply plug this lead into the TV receiver,

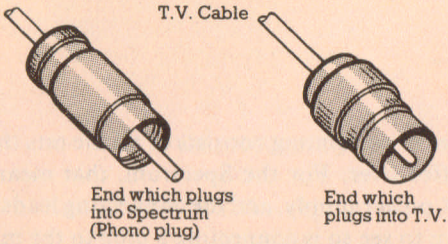


Fig. 1.2. The two different plugs on the TV cable.

but a better option is to use the type of 2-to-1 adaptor that is illustrated in Fig. 1.3. This allows you to keep an aerial cable plugged in, and to connect or disconnect the Spectrum as you wish without disturbing the TV receiver. It's useful if you have to share a colour TV with the family. It also saves wear on the aerial connector of the TV receiver itself. If you have a TV that you can reserve for use with the Spectrum then you won't need this device. The TV that you use to display the Spectrum's signals need not be a colour receiver, not to start with at least. The skills of programming a Spectrum do not require you to see the results in colour until you come to the colour instructions of the Spectrum in Chapter 7. If you use a black/white receiver, such as the little Ferguson portable which has served me so well, you will see the Spectrum colours as shades of grey.

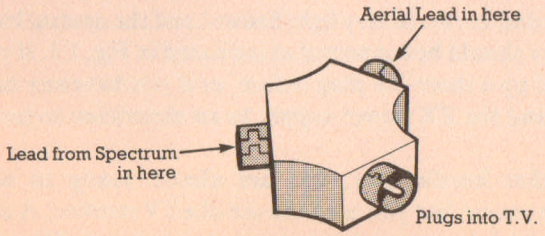


Fig. 1.3. A useful 2-to-1 TV adaptor. This is sold in TV stores as a Panda-Pack.

Socket and see

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around. When you are in full control of your Spectrum you will need three main sockets. Two of these will be for the Spectrum and the TV receiver, but you will need one more for a cassette recorder. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three- or four-way socket strip with a cable and a plug (Fig. 1.4). This avoids a lot of what the

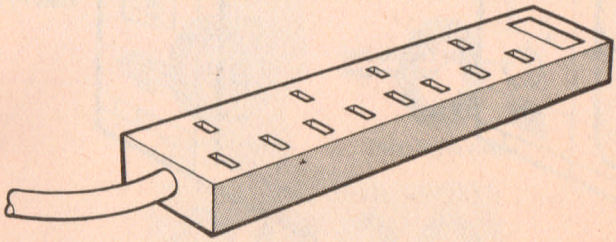


Fig. 1.4. A four-way socket strip which avoids the use of the old-style adaptors.

famous advert calls 'spaghetti hanging out the back'. Don't rely on the old-fashioned type of three-way adaptor – they never produce really reliable contacts. The Spectrum has no mains switch, so it's an advantage if you can connect it to a switched socket. If you can't, then always switch on and off by plugging and unplugging the three-pin mains plug. Never switch the Spectrum on and off by using the little jack plug which connects the ZX power supply to the Spectrum. The reason is that the small socket that this plug fits into can become loose, and when this happens, you simply won't be able to rely on the connection. When a computer is disconnected from its power supply, it instantly loses anything that was in its memory. If you have just spent an hour typing instructions on the keyboard, you won't be wildly happy if a twitch on the power-supply lead causes you to lose the lot!

The next step, then, is to switch on the TV receiver, and the Spectrum. The signals have to be transmitted, using a miniature transmitter that is called a modulator. This is because most TV receivers cannot be safely connected to anything except by the aerial lead.

The TV receiver has to be tuned to these signals from the Spectrum. Unless you have been using a video cassette recorder and the TV has a

tuning button that is marked 'VCR', it's unlikely that you will be able to get the Spectrum tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the Spectrum's signals.

Figure 1.5 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system

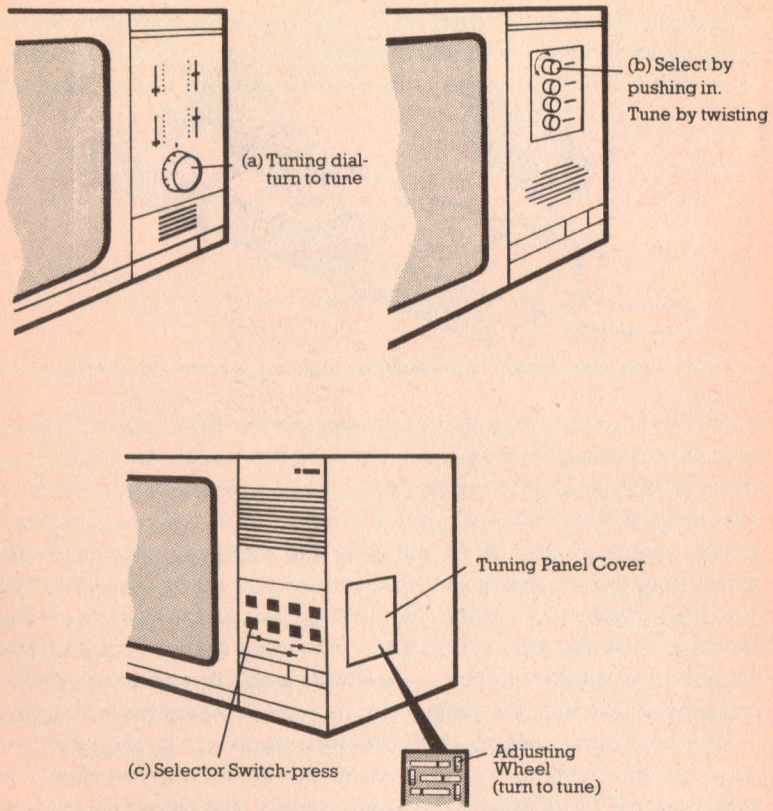


Fig. 1.5. TV tuning controls. (a) Single dial, as used on black and white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

that is illustrated in Fig. 1.5(a). This is the type of tuning system that you find on black/white portables, and to get the Spectrum's signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then

start with the dial turned fully anti-clockwise as far as it will go, and slowly turn it clockwise until you see the Spectrum signal appear.

What you are looking for, if the Spectrum hasn't been touched since you switched it on, is the phrase '© 1982 Sinclair Research Ltd.' on the screen. When you can see these words, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. On a TV receiver, particularly a colour TV, the words may never be particularly clear, but get them steady at least and as clear as possible. Figure 1.6 illustrates some faults that are caused by incorrect tuning.

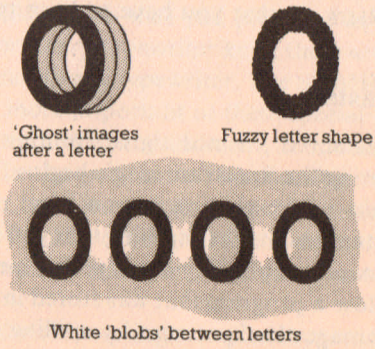


Fig. 1.6. Picture defects caused by faulty tuning.

The older types of colour and B/W TV receivers used mechanical push-buttons (Fig. 1.5(b)) which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one which for most of us means the fourth one. Push this one in fully. Tuning is now carried out by rotating this button. Try rotating anti-clockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the Spectrum's signal during this time, you'll see the same sign – the message on the screen. If you've turned the button all the way anti-clockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the Spectrum signal at any setting, check the TV using an aerial in case there is something wrong with the tuning of the TV.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the

side or at the front of the receiver (Fig. 1.5(c)). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen. On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave this cover off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep re-tuning.

Mystery and mastery

Once you have achieved a tuned signal from your Spectrum, the business of mastering the Spectrum magic begins. To start with, you have the copyright notice shining at you from the bottom of the screen. It's important to note that nothing that you can do by pressing keys on the keyboard can possibly damage the Spectrum – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the Spectrum by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. Pressing keys at random can, however, cause the computer to look as if it has 'seized up', refusing to do anything. You can always escape from these tantrums in two ways. One way is to press the keys at the opposite ends of the nearest row. They are marked CAPS SHIFT and SPACE, with the word BREAK printed above SPACE. Pressing these keys together will nearly always restore normal operation. In the very few cases in which this doesn't work, you will have to switch off and on again. By doing this, however, you will lose any program that was in the memory, so the use of the two keys is always preferable.

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the Spectrum. Most of the Spectrum keys are arranged rather like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter. If you've ever used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the Spectrum pretty quickly.

There's one very noticeable difference, though. Whereas each key

of a typewriter will give only two actions – for example, a capital letter and a small letter – each Spectrum key can give about five actions! This is done in two ways. One way is the order in which you use the keys, and we'll learn about that as we go along. The other is the use of the two SHIFT keys. One of these is the CAPS SHIFT that we have noted already, the other is the SYMBOL SHIFT which is next to the SPACE key, near the right-hand side of the first row of keys. The SYMBOL SHIFT key is marked in red because it has to be used to obtain the words and symbols that are marked in red on the other keys. Once again, we'll go into that later. As you will see, the Spectrum will guide you so that you press the keys correctly!

As well as the ordinary typewriter keys, there are two special keys which are not found on any typewriter. The SYMBOL SHIFT key is one of these. The more important of these special keys, however, as far as we are concerned at the moment, is the key that is marked ENTER. This is in the position of the 'carriage return' key of an electric typewriter, but its action is not the same in all respects. Pressing the ENTER key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it.

If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the 'carriage return' key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The ENTER key of the computer does rather more than this. If the material that you are typing into the Spectrum takes more than one line on the screen, the machine will automatically select the next screen line for you. The ENTER key must not be used for this purpose. The ENTER key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by a flashing letter on the bottom line of the screen. This flashing letter is called the 'cursor', and it acts as a sort of signpost for you, as we'll see later.

Save it!

You can obtain a lot of enjoyment from a computer system that

consists only of the machine and a TV receiver. Each time that you switch the machine off, however, all the program and other information that has been stored in the memory of the computer will be lost. Since it might take several hours to enter a program into the machine by typing instructions on the keyboard, this waste just has to be avoided. We avoid the loss of programs by recording them on tape. Before you tackle the rest of this book, then, it's important to check now that you can record and replay programs.

Almost any cassette recorder is suitable, but if you are buying a recorder specially for computer use, get one which is mains or battery operated, has a tape counter and automatic recording level control. Most small portable cassette recorders have these features, but hi-fi and other stereo recorders are usually unsuitable. I have used a Trophy CR100 recorder for years with very satisfactory results, and Boots now sell an identical machine, Boots CR325. Recorders by Sanyo, Hitachi and Sharp have also proved satisfactory. If you are buying a cassette recorder to use with your Spectrum, it's a good idea to ask the shop to demonstrate it being used along with a Spectrum, because not all cassette recorders work equally well.

Start work by switching everything off. Now find the cassette lead of the Spectrum. This has two small plugs at each end. These small plugs, which are called jack-plugs, are colour-coded. The black plug which is fitted at one end of the lead engages into the socket on the cassette recorder that is marked MIC or which has a drawing of a microphone. The black plug at the other end of the lead fits into the socket which is marked MIC on the back of the SPECTRUM. The other plug, coloured grey, fits into the EAR socket on the Spectrum, and will later fit into the EAR socket of the recorder. This socket may be marked with a drawing of an ear. Don't, however, push this plug into the recorder at the moment. Try to cultivate the habit of leaving it out unless you are replaying a cassette. The Spectrum uses only these two plugs.

Once you have made the connections between the MIC sockets, the cassette recorder is ready for use. It's preferable to run the recorder from the mains because battery life can be unpredictable, and if your batteries decide to fail while you are recording, you will probably lose the program. The next thing that you have to sort out is a supply of blank cassettes. There's nothing wrong with using reputable brands of C90 length cassettes (ordinary 'ferric' tape, not the hi-fi CrO_2 type), but you'll find that the short lengths of tape that are sold as C5, C10 or

C15 in computer shops and in most branches of Boots, W. H. Smiths and Currys are much more useful.

Put a fresh cassette into the machine, with the 1 or A side uppermost. The first part of the cassette consists of a 'leader', which is plain, not recording, tape. This has to be wound on before you can record. If your recorder has a tape counter, reset the counter to zero, and then fast-wind the cassette to a count of 5. If there is no tape counter, take the cassette out and insert the body of a BiC pen into the centre of the empty reel. Turn the pen so that the tape winds on to the reel, and keep turning until you see the brown recording tape replace the clear or brightly coloured leader.

Now before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched the Spectrum on, but if you have been pressing keys at random, then it's a good idea to switch off again, then on. Press the ENTER key when you see the copyright notice appear, and your Spectrum is ready for testing.

Type the number 1Ø (1 and then Ø), and then press the 'E' key. This will place the word REM on the bottom line of the screen, just following the 1Ø. Check that this looks correct, and then press the ENTER key. The effect of this is to place the instruction line 1Ø REM into the memory of the Spectrum. Now type the rest of the lines, as illustrated in Fig. 1.7, remembering to press the ENTER key after you have completed typing each line. The numbers are called 'line

```
10 REM
20 REM
30 REM
40 REM
```

Fig. 1.7. A program for testing the cassette recording and replaying actions.

numbers', and they are there for two reasons. One is to remind the computer that this is a program, the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers. Following a line number, the computer must have an instruction word, and that's why the word REM appears rather than the letter 'E' when you press the E key.

Check that your program looks on the screen like the printed version in Fig. 1.7, and make sure that the cassette recorder is ready. Now press the S key, which will produce the instruction word SAVE on the screen. SAVE is the instruction to the computer meaning that

you want to save (record) a program on a cassette. This action will also cause the flashing letter to change to an 'L'. This won't do by itself, however. You now have to put in quote marks (inverted commas). This is done by holding down the SYMBOL SHIFT key and, while this is held down, pressing the 'P' key. You can see the quote marks on the P key in red. Now press the 'a' key, to obtain a letter 'a'. The "a" is a *filename* which the computer will use to recognise the program if it is asked to. You must put in another quote mark (inverted commas, obtained by pressing SYMBOL SHIFT and P at the same time), otherwise the computer cannot carry out the instruction. Press ENTER now, and you will see the message 'Start tape, then press any key' appear on the bottom line of the screen. Now start the recorder by pressing its PLAY and RECORD keys. Press them firmly so that they lock in place, and you will see the reels of the cassette turning. When the recorder is working, press the ENTER key on the Spectrum – you can, in fact, press any key. You will then see the centre of the screen clear, and a set of bands appear round the borders. If you are using a colour TV you will see that these bands are coloured. There are two lots of bands, so don't be tempted to switch off the recorder too soon. After a very short time, the cursor of the Spectrum will reappear on the screen with a '0 OK, 0:1' message. This lets you know that the program has been recorded, and you can switch the recorder off – that's all. Now set the volume control of the recorder to half-way along its range, and the tone control, if any, to maximum treble.

Now comes the crunch. You have to be sure that the recording was O.K. The first thing to do is to plug the EAR connection into the recorder. Wind back the tape, using the rewind key of the recorder, and type:

VERIFY

You get this by holding down the CAPS SHIFT key, and tapping first the SYMBOL SHIFT key and then the 'R' key in turn while CAPS SHIFT is held down. When you see the word on the screen, press the SYMBOL SHIFT key along with the 'P' key to get the quote mark, and then press the 'a' key by itself to get the letter 'a'. Then press SYMBOL SHIFT and 'P' to get the other quote mark. Now press ENTER this command will cause the Spectrum to compare what is stored in its memory with the program that you recorded. It can't do so, however, until you play the program back.

Press the PLAY key of the recorder, and wait. You will see the border of the screen flash slowly. After a time, you should see the screen clear and the border display moving stripes again. When the screen shows the message '∅ OK, ∅:1' you can stop the recorder. Your cassette recording is O.K., and correct recordings are being made. Just to show that the program has indeed been recorded, wind back the tape again. Type NEW by pressing the 'A' key and press ENTER. This should have wiped your program from the memory. Now type LIST by pressing the 'K' key and press ENTER. Nothing should appear – LIST means put a list of the program instructions on the screen, and there shouldn't be any!

You can now load the instructions in from the tape. Type LOAD by pressing the 'J' key, and then put in "a" in the same way as before, and press ENTER. Now press the PLAY key of the recorder (did you rewind the tape?). The stripes will appear to show you that the program is loading, and the message will show when the loading operation is complete. When this appears, the program is in place, and the recorder can be stopped. Type LIST now ('K' key), then press the ENTER key. You should see your program appear on the screen.

Once you can reliably save programs on tape, verify them, and re-load them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes more work will save your effort on tape so that you won't have to type it again.

What can go wrong? Bad connections, mainly, and the Table in Fig. 1.8 should help you to trace the source of problems. The essential thing to remember is that the EAR plug must be disconnected while you are recording a program. One other problem remains, however, which you may find puzzling. Even though you can record and replay your own programs, you may find that when you buy a program on a cassette it refuses to load at any setting of the recorder's volume control. This is nearly always because the head of the cassette recorder needs attention and there's advice on this problem in Appendix A. Try loading the 'HORIZONS' tape that comes with your Spectrum. This tape contains a check for correct volume control level, and the first parts of the tape are concerned with checking that this is set. If you can load this tape with no problems, then your recorder is O.K. If you cannot load this tape, an audio dealer should be able to adjust the tape recorder for you.

Error Check List

Note. Computer manuals always state that any cassette recorder can be used along with computers. A few recorders, however, have proved to be completely unsatisfactory, because they are unable to cope with the strong signals that the Spectrum (and other computers) send out. Unfortunately, some stores sell as suitable for computer use a recorder that appears to cause considerable trouble, unless it is modified. If you buy a recorder specially to use with your Spectrum, you should insist on seeing it save and load a program first.

1. Check first of all to find if anything has been recorded on your cassette. Pull out both plugs from the recorder, rewind the cassette, and play it, with the volume control about quarter way up. Listen to the tape. If you hear a shrill rasping noise, quite loud, then the program has probably recorded. If there is no sound, or just a low-pitched hum, then it hasn't recorded – so go to 6.

2. If the program was recorded but did not VERIFY correctly or LOAD again, you may have recorded it with the EAR plug in place. Try a new recording, making sure that the EAR plug is out when you make the recording. Allow a few seconds between pressing the PLAY and RECORD keys on the recorder and pressing any Spectrum key to start the recording.

3. Make sure that the EAR plug is in place when you VERIFY or LOAD. The setting of the volume control may be critical on your recorder. Try a range of settings, altering the control only slightly between attempts. Use the CAPS SHIFT and SPACE keys to end a LOAD attempt before you try with another setting. Once you have achieved the correct setting, mark it!

4. Make sure that the plugs are correctly connected at each end.

5. Don't try to record over a previous recording. If you really want to re-use cassettes, clear them by using a 'bulk eraser'. Use a reliable brand of recording tape, the ordinary 'ferric' variety rather than the hi-fi 'chrome' type. Don't use long lengths of tape.

6. Make sure that the recorder is capable of recording, by using its microphone to record a few spoken words, and then play them back.

7. If all else fails, have your recording leads checked by a Spectrum dealer. If these seem all right, borrow another cassette recorder. Once you have exhausted all of these possibilities, the only remaining item is to return the Spectrum and request it to be checked by the dealer.

Fig. 1.8. A check list for recording faults.

Chapter 2
Screen Messages

Chapter 1 will have broken you in to the idea that the Spectrum, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the ENTER key is pressed. You will have used the command NEW which clears out a program from the memory; and LIST which prints your program instructions on to the screen. Now there are two ways in which you can use a computer. One way is called *direct mode*. Direct mode means that you type a command, press ENTER, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*.

The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press ENTER.

Let's take a look at the difference. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

PRINT 1.6 + 3.2 (and then press ENTER)

You have to start with PRINT because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like 'GIVE ME' or 'WHAT IS' – only a few words that we call its *reserved words* or *instruction words*. PRINT is one of these words. The PRINT instruction is obtained by pressing the 'P' key when the

flashing letter is a 'K'. Remember to press the SYMBOL SHIFT key to obtain the decimal point and the + symbols. When you press ENTER after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8, at the top, and the message '0 OK. 0:1' at the bottom. Once this command has been carried out, however, it's finished.

A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press ENTER. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your commands and your program instructions. This is done by starting each program instruction with a positive whole number which is called a line number. This is why you can't expect the computer to understand an instruction like 5.6 + 3= ; it takes the 5 as being a line number, and the rest doesn't make sense.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

```
10 PRINT "5.4+3.2= ";5.4+3.2
20 PRINT "1.7*4.2= ";1.7*4.2
30 PRINT "12.6/2.4= ";12.6/2.4
40 PRINT "9.7-6.2= ";9.7-6.2
```

Fig. 2.1. A four-line arithmetic program.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 10,20,30,40 rather than 1,2,3,4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 10 and line 20, then you can type the line number 15, or 11 or 12 or any other whole number between 10 and 20, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 10 and 20. If you number your lines 1,2,3 then there's no room for these second thoughts.

The next thing to notice is how the number zero is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0, and the slashing makes this difference more obvious to you, so that you are less likely

to make mistakes. Some magazines, unfortunately, reprint computer programs with the slashmarks removed, so that it's very easy to make mistakes.

Now to more important points. The star or asterisk symbol in line 2Ø is the symbol that the Spectrum uses as a multiply sign. Once again, we can't use the × that you might normally use for writing multiplication because '×' is a letter. There's no divide sign on the keyboard, so the Spectrum, like all other small computers, uses the backslash (/) sign in its place.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the Spectrum will put one in for you. You will have to press the ENTER key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration.

When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. One is how to check that the program is actually in the memory, the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use CLS (on the V key), followed by pressing ENTER to wipe the screen first if you like, then type LIST and press the ENTER key. When you press the ENTER key, and not until, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. To make the program operate, you need another command, RUN. Type RUN, then press the ENTER key, and you will see the instructions carried out. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
5.4+3.2=8.6
```

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "5.4+3.2= ";2+2
```

then you'll get the daft reply, when you RUN this, of:

5.4+3.2= 4

The computer does as it's told and that's what you told it to do. Screwdrivers haven't taken over the world, so why should computers!

This is a good point also to take notice of something else. The line 15 that you added has been fitted into place between lines 1Ø and 2Ø – LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of ascending line number for you. Note also that the spaces in the program of Fig. 2.1 between the = and the " are useful – just see what happens if you miss them out!

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, as far as the Spectrum is concerned, always means print on to the TV screen. For activating the ZX printer, there's a separate instruction LPRINT (and LLIST for program listings). You must not use these instructions unless you have a printer connected and switched on.

Now try the program in Fig. 2.2. You can try typing the lines in any

```
10 PRINT "This is Spectrum"
20 PRINT "your entry";
30 PRINT " to the world of com
puting."
40 PRINT ''
50 PRINT " _ in colour."
```

Fig. 2.2. Using the PRINT instruction to place words on the screen.

order that you like, to establish the point that they will be in line number order when you list the program. When you RUN the program, the words – This is Spectrum – appear on one line, and the words – your entry to the world of computing – on the next line. This is because the instruction PRINT doesn't just mean 'print on the screen'. It also means 'take a new line', and start at the left-hand side!

Now this isn't always convenient, and we can change the action by using punctuation marks that we call *print modifiers*. In line 2Ø, for example, the semicolon at the end of the line will cause printing to continue on the same line. You have to be careful how you do this, because you will jam words together if you don't leave a space, and you also have to watch how words are split up at the end of a line. The Spectrum allows you 32 characters in each printed line, which is why

part of the word 'computing' has been printed one line down. Line 4Ø shows a way of arranging typing more neatly. The apostrophe mark (') causes printing to skip one line. We'll look at examples of this later being used to split up long phrases so that they fit more neatly on the screen. For the moment, though, we have another printing trick to learn.

Start this time by acquiring a new habit. Type NEW (on the 'A' key) and then press the ENTER key. This clears the old program out. If you don't do this, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored.

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. Figure 2.3 shows one way of arranging columns. Line 1Ø is a

```
10 PRINT "Number","Colour"
20 PRINT "0","black"
30 PRINT "1","Blue"
40 PRINT "2","Red","3","Purple
"
```

Fig. 2.3. How the comma causes words to be placed in columns.

PRINT instruction that acts on the words 'Number' and 'Colour'. Note that you get the capital letters for N and C by pressing CAPS SHIFT at the same time as you press the letter key, just as on a typewriter. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into two columns. The mark which causes this effect is the comma, and the action is completely automatic. As line 4Ø shows, you can't get more than two columns. Anything that you try to get into a third column will actually appear on the first column of the next line down. The action works for numbers as well as for words, as lines 2Ø to 4Ø illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed outside the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into columns something that is too large to fit, the long

phrases will spill over to the next column, and the next item to be printed will be at the start of the next column along.

Commas are useful when we want a simple way of creating two columns. A much more flexible method of placing words along a line exists, however, and is illustrated in Fig. 2.4. It uses the instruction

```
10 PRINT ;"Side"
20 PRINT TAB 8;"tab8"
30 PRINT TAB 20;"tab20"
40 PRINT TAB 25;"side"
50 PRINT TAB 10;"ten";TAB 20;2
0
```

Fig. 2.4. Tabulation in a program.

word TAB, which has to be followed by a number. The command word TAB is obtained by pressing both of the SHIFT keys, releasing them, and then pressing the 'P' key. The flashing letter changes to an 'E' when you have pressed both SHIFT keys, and this is the method that you have to use to enter any of the words or symbols that are printed in green above the keys. To see what TAB does, imagine the line divided into 32 portions, equally spaced, and numbered from 0 to 31. Thus TAB 0 means the position at the left-hand side of the line, and TAB 31 means the position at the right-hand side of the line. Figure 2.4 illustrates the appearance of words that are typed at different TAB positions. Note that we must use TAB only following a PRINT instruction, but we can use more than one TAB following a PRINT instruction, as line 50 demonstrates.

The use of *tabulation*, as this is called, can make the appearance of printing on the screen much smarter, as Fig. 2.5 illustrates. In this

```
10 CLS : PRINT TAB 13;"TITLE"
20 PRINT ''
30 PRINT TAB 2;"Text looks nea
ter"
```

Fig. 2.5. Using TAB to make printing look neater.

example, the word TITLE has been centred on its line by using TAB 13. Line 10 contains a novelty, though, in the form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, even if they require several screen lines to type. In a 'multistatement' line of this type, the Spectrum will deal with the different instructions in a left-to-right order. Getting back to printing

the word centred on the line, however, the TAB number is found by using a formula that has been known to typists for generations – it's illustrated in Fig. 2.6. Later on, we'll look at ways of carrying out this

```
Spectrum Independent
20 characters
(count space between m and I)
32 characters per line
32 – 20 = 12
½ of 12 = 6
so use TAB 6
PRINT TAB 6; "SPECTRUM INDEPENDENT"
```

Fig. 2.6. The formula for centring a title.

calculation automatically, so that you can print any phrase centred in a line without having to count the letters and spaces for yourself. One point that we haven't illustrated yet is that the quantity that follows a TAB need not be a number – it can be a letter that represents a number. We'll come back to that point in the next chapter.

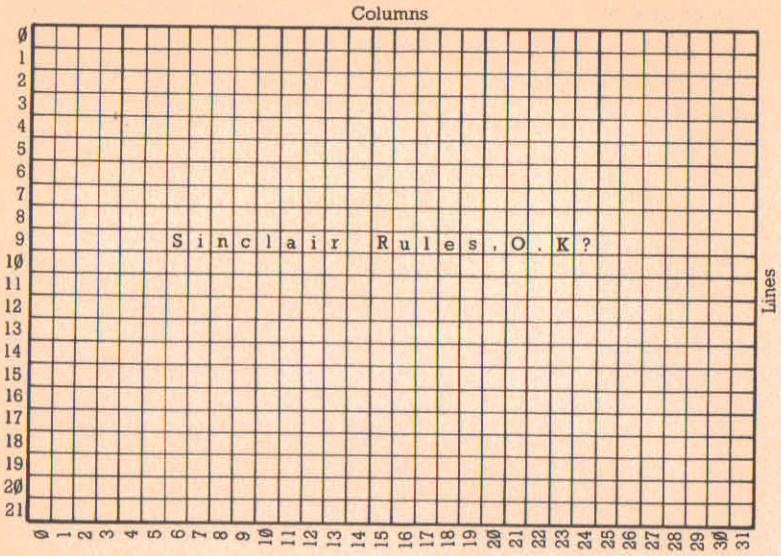
Take a look now at the use of TAB for forming columns on the screen. Figure 2.7 illustrates different TAB positions being used to produce four columns. It also shows how we have to use semicolons

```
10 CLS : PRINT TAB 12;"COLUMNS
"
  20 PRINT TAB 7;"A";TAB 14;"B";
TAB 21;"C";TAB 28;"D"
  30 PRINT ': PRINT TAB 7;1;TAB
14;2;TAB 21;3;TAB 28;4
```

Fig. 2.7. Using TAB to produce columns.

as separators – you must not omit the semicolon that follows the TAB number. If you press ENTER, having forgotten the semicolon, the Spectrum will remind you of the error by flashing a question mark in the place where you should have put the semicolon. You can get back to the place by deleting – using CAPS SHIFT and the ∅ key. An alternative is to use the CAPS SHIFT with the left-arrow key (the '5' key) – this method is not fully described in the manual.

Meantime, there's another very important print modifier to look at. The AT word on the keyboard is used to allow text (numbers, letters, words) to be placed anywhere on the screen. For the purpose of using AT, we imagine the screen divided into a grid of 32 divisions across and 22 lines down, as Fig. 2.8 shows. Instead of the single number which we use to follow TAB, AT is followed by two numbers. The first of these numbers is the line number, counting from the top with the top line numbered zero. The second number is the same



(Bottom lines 22, 23, are reserved for inputs, remember.)
The example shows the result of the command
PRINT AT 9, 6; "Sinclair Rules, O.K?"

Fig. 2.8. The PRINT AT grid.

number as we would use for a TAB, the number of spaces along the line, counting the left-hand side as position zero.

Figure 2.9 shows how PRINT AT is used. You must separate the

```
10 CLS : PRINT AT 3,13;"TITLE"
20 PRINT AT 20,13;"first"
30 PRINT AT 10,13;"second"
40 PRINT AT 1,8;"third";AT 1,2
0;"fourth"
```

Fig. 2.9. How PRINT AT is used to place words anywhere on the screen.

numbers with a comma, and you have to place a semicolon between the last number and the item that you want to print. You can use more than one AT following a PRINT in a line. This is illustrated in line 40 – you have to remember to separate the next AT by using a semicolon. The effect of PRINT AT is to allow you to print items wherever you want, as Fig. 2.9 shows. You don't have to print in the order of left to right or top to bottom either, because PRINT AT allows you complete freedom to print wherever you want. If your

choice of PRINT AT position places a new word over an old one, then the new letters will simply replace the old ones.

As a final point, we saw earlier that the Spectrum will print small (*lower-case*) letters normally and will print capitals (*upper-case*) if you type a letter while you are holding down the CAPS SHIFT key. You can type in all capitals by using the CAPS LOCK action, which is obtained by pressing the CAPS SHIFT key along with the '2' key. This works just like the SHIFT LOCK of a typewriter, causing all the letters that you type to be in capitals. This applies only to letters, however, when the flashing cursor is an 'L'. The other actions of the keys are unaffected.

Chapter 3
Spice Of Life?

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. One of these is called *assignment*.

Take a look at the program in Fig. 3.1. Type it in, run it, and

```
10 CLS
20 LET x=15
30 PRINT "2 times ";x;" is ";2
*x
40 LET x=20
50 PRINT "x is now ";x
60 PRINT "and 2 times ";x;" is
";2*x
```

Fig. 3.1. Introducing the use of a variable name.

contrast what you see on the screen with what appears in the program. The first line that is printed is line 3Ø. What appears on the screen is:

2 times 15 is 3Ø

but the numbers 15 and 3Ø don't appear in line 3Ø! This is because of the way we have used the letter x as a kind of code for the number 15. The official name for this type of code is *variable name*.

Line 2Ø assigns the variable name x, giving it the value of 15. 'Assigns' means that wherever we use x, not enclosed by quotes, the computer will operate with the number 15. Since x is a single character and 15 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned x differently, perhaps as LET x=2174.3256, for example. Line 3Ø then proves that x is taken to be 15, because wherever x appears, not between quotes, 15 is printed, and the 'expression' 2*x is printed as 3Ø. We're not stuck

with x as representing 15 for ever, though. Line 4Ø assigns x as being 2Ø, and lines 5Ø and 6Ø prove that this change has been made.

That's why we call x a 'variable' – we can vary whatever it is we want it to represent. Until we do change it, though, x stays assigned. Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type PRINT x, and pressing ENTER will show the value of x on the screen.

This very useful way to handle numbers in code form can use a 'name' which must start with a letter. You can add to that more letters or numbers, so that n,r2d2 or number, for instance, are all names that you can use for number variables, and each can be assigned to a different number. Just to make it even more useful, you can use code 'names' to represent words and phrases also. The difference is that these 'names' must consist of one letter only, and you have to add a dollar sign ($) to the variable name. If n is a variable name for a number, then n$ (pronounced 'en-string' or 'en-dollar') is a variable name for a word or phrase. The computer treats these two, n and n$, as being entirely separate and different.

Figure 3.2 illustrates 'string variables', meaning the use of variable names for words and phrases. Lines 1Ø and 2Ø carry out the

```
10 CLS : LET s$="Spectrum"
20 LET a$="The ZX "
30 PRINT AT 2,13;a$
40 PRINT AT 4,12;s$
50 PRINT "This is ";s$
```

Fig. 3.2. Using string variables. These are distinguished by the dollar sign.

assignment operations, and lines 3Ø and 5Ø show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which must be surrounded by quotes.

Figure 3.3 shows another example of the use of variable names. This time the variable names are contrasted – the number uses a long variable name, and the string uses the compulsory single letter. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a phrase

```
10 CLS : LET pettycash=24
20 LET a$="Jones account"
30 PRINT a$;" petty cash is £"
;pettycash
```

Fig. 3.3. String and number variables being used together.

in a program, this is one method of programming it so that you don't have to keep typing it! Another point about variables is that the use of a long variable name can make it easier for you to remember what it is that a name is supposed to represent.

String along with me

Because the name of a string variable is marked by the use of the $ sign, a variable like a$ is not confused with a number variable like a. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the difference is a bit more than skin deep, though. Lines 1Ø and 2Ø assign

```
10 CLS : LET a=12: LET b=3.7
20 LET a$="12": LET b$="3.7"
30 PRINT a,b
40 PRINT a$,b$
50 PRINT "a*b is ";a*b
60 PRINT "but a$*b$ is impossi
ble!"
```

Fig. 3.4. Strings and numbers might look alike, but they are different!

number variables a and b, and string variables a$ and b$. When these variables are printed in lines 3Ø and 4Ø, you can't tell the difference between a and a$ or between b and b$. The difference appears, however, in line 6Ø. It can multiply two number variables as we've done in line 5Ø, because numbers can be multiplied, but it can't multiply string variables. The reason is simple. A string variable can be anything. We have assigned a$ as '12', but we could just as easily have assigned it as '12 LAVENDER WAY'. You can multiply 12 by 3.7, but you can't multiply 12 LAVENDER WAY by 3 ORCHID AVENUE. The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to program a forbidden operation in line 6Ø would cause an error message. The Spectrum catches errors like this before you can enter them into the memory, and you will see the question mark flashing just after the first string sign if you attempt to enter:

```
6Ø PRINT a$*b$
```

The operation that we are calling for can be done on numbers, but we have strings here. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and

attempts to do these operations on numbers will also cause an error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's wonderful, but it's only a machine!

There is one operation, that looks rather like arithmetic being carried out on strings. It uses the + sign, but it isn't addition in the sense of adding numbers. Figure 3.5 illustrates this action of joining strings, which is often called *concatenation*. This is nothing like the

```
10 CLS : LET a$="The ZX"
20 LET b$=" SPECTRUM"
30 LET c$=a$+b$
40 PRINT c$
```

Fig. 3.5. Concatenating or joining strings.

action of arithmetic, as you'll see if you use numbers in place of the names. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of typing. Take a look at line 1Ø of Fig. 3.6. This defines string a$ as a set of characters which can be used as a 'frame' around a title. There is a piece of new programming here

```
10 CLS : LET a$="▚▚▚▚▚"
20 LET s$="SPECTRUM"
30 LET c$=a$+s$+a$
40 PRINT AT 2,9;c$
```

Fig. 3.6. Using concatenation to make a frame for a title.

in the way that we produce the shapes. These shapes are called *graphics*, and they will be produced after you have pressed the CAPS SHIFT and the '9' key together. Having done this, press the '6' key five times, and you will have the set of chequers. Go back to normal programming by pressing CAPS SHIFT and '9' again, so that you can put in the final quotes. The title is defined in line 2Ø as SPECTRUM. Line 4Ø then prints a concatenated string. This has needed less typing than if you had to type all the characters between the quotes. It also allows you to rearrange the frames as you please. You can, for example, define another set of graphics shapes as b$, and then use:

b$ + a$ + s$ + a$ + b$

next time you print the title.

What goes in ...

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT. Logically enough, it's on the 'I' key of your Spectrum.

Figure 3.7 illustrates this with a program that prints your name.

```
  10 CLS : INPUT "What is your n
ame?";n$
  20 PRINT "";n$;", this is your
life!"
```

Fig. 3.7. Using the INPUT instruction. The name that you type appears on the bottom line.

Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

What is your name?

are printed on the last line of the screen. The computer is now waiting for you to type something, and then press ENTER. Until the ENTER key is pressed, the program will hang up at line 1Ø, waiting for you. If you're honest, you will type your own name and then press ENTER. You don't have to put quotes around your name, simply type it in the form that you want to see printed. When you press ENTER, your name is assigned to the variable n$. The program can then continue, so that line 2Ø then prints the famous phrase with your name at the start. Take a close look at this – the words that you want to see printed must be enclosed by quotes, and you have to separate the words from the variable name by using a semicolon. If you try to ignore these rules, the Spectrum simply won't allow you to enter the line.

You could, of course, have answered MICKEY MOUSE or DONALD DUCK or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.8 illustrates an INPUT step which uses a number variable age and a string variable n$. Now when the computer comes to line 1Ø, it will print the message and then wait for

```
  10 CLS : INPUT "Name and age,P
lease";n$,age
  20 PRINT "Well ,";n$;" ,I supp
ose you are";"looking forward to
 being ";age+1
```

Fig. 3.8. Putting in two variables in one INPUT step.

you to enter both of these quantities, a name and then a number. You have to be particular about how you enter these quantities. If you type the name and then press ENTER, the computer will print its flashing cursor (the 'L' shape) immediately following your name on the bottom line of the screen. This is its way of indicating 'more needed', and that's a signal for you to type the number and then press ENTER again. The name and number will be printed again in line 2Ø. There's one more thing to watch. When you use a number variable in an INPUT step, then what you have typed when you press ENTER must be a number. If you attempt to enter a string, the computer will refuse to accept it, and you will see the error message '2 variable not found, 1Ø:2' appearing. This is the signal to you that something has gone wrong – and the problem is in the second part of line 1Ø. The trouble with this is that it stops the program dead. Later on we'll look at ways of avoiding this problem by making every entry a string entry.

How many beans make n?

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for accounts or for word processing. It's time, then, to take a very brief look at the number abilities of the Spectrum. It is a brief look because we simply don't have space to explain what all the mathematical operations do. In general, if you understand what a mathematical term like *sin* or *tan* or *exp* means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of *incrementing* if you are counting up and *decrementing* if you are counting down. Incrementing a number

means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in BASIC, as Fig. 3.9 shows. Line 1Ø sets the value of the variable

```
  10 CLS : LET no=12
  20 PRINT "number is ";no: LET
no=no+1
  30 PRINT "now it's ";no
  40 PRINT "its  square is ";no↑
2
  50 PRINT "its  square root is
";SQR no
```

Fig. 3.9. Incrementing, using the equals sign to mean 'becomes'.

number as 12. This is printed in the first part of line 2Ø, but then the second part of line 2Ø 'increments no'. This is done using the odd-looking instruction: LET no = no + 1, meaning that the new value that is assigned to no is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of no has been carried out.

The use of the = sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

LET no= no−1

This would have the effect of making the new value of 'no' one less than the old value; 'no' has been decremented this time. We could also use LET no = 2*no to produce a new value of 'no' equal to double the old value, or LET no = no/3 to produce a new value of 'no' equal to the old value divided by three.

Number functions

Figure 3.9 also illustrates some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 2Ø has changed the value of 'no' to 13. Line 4Ø then prints the value of 'no' squared, (meaning 'no' multiplied by 'no'). This is programmed by typing no 12, with the up-arrow on the 'H' key. To get the square root of the number that has been assigned to 'no', we use the instruction word SQR, which is above the 'H' key. Remember how to get one of these words that are printed in green. You press both shift keys together, release them both, then

press the 'H' key, in this example. An alternative is no↑.5, but 'SQR no' is easier to type and remember. For other roots, like the cube root you can use expressions like no↑(1/3) and so on. Notice the use of the brackets – you can't leave them out, otherwise the computer will treat the instruction as being no to the power of 1, then divided by 3, rather than no to the power of one-third.

Figure 3.10 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of interest only if you want to program for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs.

Function	Followed by	Action
ABS	Number	Size, ignoring sign
ACS	Number	Arc (or inverse) cosine
ASN	Number	Arc (or inverse) sine
ATN	Number	Arc (or inverse) tangent
COS	Number	Cosine of angle
EXP	Number	Exponential of number
FN	Letter	User-defined function
INT	Number	Rounds number down to whole number
LN	Number	Natural logarithm
PI	Nothing	Gives value of pi
RND	Nothing	Random value between ∅ and 1
SGN	Number	Gives sign of number
SIN	Number	Gives sine of number
SQR	Number	Gives square root of number
TAN	Number	Gives tangent of number

Note: The trigonometrical functions, SIN,COS and TAN require a number which is the angle in radians. (2π radians = 360°. Thus, 1 radian = 57.2958°.) The inverse functions ASN and ACS will accept numbers between −1 and +1 only.

Fig. 3.10. Spectrum number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

How precise?

One of the problems of small computers is precision of numbers. You

probably know that the fraction 1/3 cannot be expressed exactly as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that 0.33 is closer than 0.3, and 0.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a *binary fraction*, and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 10,100 and also .1, .01, .001; all the powers of ten, in fact. To avoid embarrassments like printing 3 − 2 = .9999999, the computer will round numbers of this type up or down as need be before displaying them. You will sometimes find, however, that a piece of arithmetic looks spectacularly wrong, simply because it has used numbers that cannot be represented exactly inside the computer. As an example, try:

```
PRINT 10-100*.1
```

and see what you get. This is not a fault of Spectrum, it's a problem that exists in all computers, and you have to find ways around it if you want to work with precise numbers.

Chapter 4
Repeating The Process

Every computer is well equipped with instructions that will cause repetition, and the Spectrum is no exception. We'll start with the simplest of these 'repeater' actions, GOTO. GOTO means exactly what you would expect it to mean – go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again.

Figure 4.1 shows an example of a very simple repetition or *loop*, as we call it. Line 1Ø starts the program by assigning a variable n to have

```
  10 LET n=1
  20 PRINT "SPECTRUM----------
---SPECTRUM"
  30 PRINT n
  40 LET n=n+1: GO TO 20
  50 REM Press N key to stopwhen
 you see the "scroll?" question
appear, or use CAPS SHIFT and SP
ACE
```

Fig. 4.1. A very simple loop. You can 'break' this either by pressing 'n' when the 'scroll' question is asked, or by pressing CAPS SHIFT and SPACE together.

the value of 1. Line 2Ø contains a simple PRINT instruction. When line 2Ø has been carried out, the program moves on to lines 3Ø and 4Ø, which print the value of variable n, and increase it by one. The last part of line 4Ø then instructs the program to go back to line 2Ø again. This is a never-ending loop, and it will cause the screen to fill with the word SPECTRUM and the number until you press the CAPS SHIFT

and SPACE keys to 'break the loop'. Any loop that appears to be running forever can normally be stopped by pressing these keys.

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! We don't always have an alternative, but there is one – the FOR...NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instructions that are repeated are the instructions that are placed between FOR and NEXT. Figure 4.2 illustrates a very simple example of the FOR...NEXT loop in action.

```
10 CLS : FOR n=1 TO 10
20 PRINT n;" Spectrum magic!"
30 NEXT n
```

Fig. 4.2. Using the FOR...NEXT loop for a counted number of repetitions.

The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, n is the counter variable, and its limit numbers are 1 and 10. The NEXT n is in line 30, and so anything between lines 10 and 30 will be repeated.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print the number and 'Spectrum magic!' ten times. At the first pass through the loop, the value of n is set to 1, and the phrase is printed. When the NEXT n instruction is encountered, the computer increments the value of n, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then line 20 is repeated, and this will continue until the value of n exceeds 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

Even at this stage it's possible to see how useful this FOR...NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR n=1 TO 9 STEP 2
```

which would cause the values of n to change in the sequence 1,3,5,7,9. When we don't type STEP, the loop will always use increments of +1. You don't have to confine this action to single loops either. Figure 4.3

```
10 CLS
20 FOR n=10 TO 0 STEP -1
30 PRINT AT 8,2;n;" seconds an
d counting."
40 FOR j=1 TO 250: NEXT j
50 NEXT n: PRINT AT 10,12;"BLA
STOFF"
60 PRINT " n is now ";n
```

Fig. 4.3. A program that uses nested loops, with one loop inside another.

shows an example of what we call *nested loops*, meaning that one loop is contained completely inside another one.

When loops are nested in this way, we can describe the loops as *inner* and *outer*. The outer loop starts in line 2Ø, using variable n which goes from 1Ø to Ø in value. Line 3Ø is part of this outer loop, printing the value that the counter variable n has reached. Line 4Ø, however, is another loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable j, and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The overall effect, then, is to show a count-down on the screen, slowly enough for you to see the changes, and printing in the same place each time. Note carefully that you have to end each loop correctly with a NEXT j and a NEXT n, according to which is required.

Every now and again, when we are using loops, we find that we need to use the value of n (or whatever name we have used) after the loop has finished. It's important to know what this will be, however, and line 6Ø brings it home to you. This reveals that the value of n is −1 in line 6Ø, after completing the FOR n = 1Ø TO Ø STEP −1. If you want to make use of the value of n, or whatever variable name you have selected to use, you will have to remember that it will have changed by one more step at the end of the loop.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 4.4 illustrates this in a simple way.

```
10 CLS : LET a=2: LET b=5: LET
c=10
20 FOR n=a TO b STEP b/c
30 PRINT n: NEXT n
```

Fig. 4.4. A loop instruction that is formed with number variables.

The letters a, b and c are assigned as numbers in the usual way in line 1Ø, but they are then used in a FOR...NEXT loop in line 2Ø. The limits are set by a and b, step is obtained from an expression, b/c. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this.

Decision steps

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done up till now, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set, because this would mean starting with a FOR n=1 TO 1Ø loop. The trouble is, how many times would you want to have just ten numbers? It would be a lot more convenient if we could stop the action simply by signalling to the computer in some way, perhaps by entering a value like Ø or 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of Ø is very convenient, because if it gets added to the total it won't make any difference.

Figure 4.5, therefore, shows an example of this type of program in action. We can't use a FOR...NEXT loop, because we don't know in

```
  10 CLS : PRINT TAB 10;"Total f
inder"
  20 PRINT "The program will fin
d the total"'"of numbers that yo
u enter "'"(using ENTER) until y
ou enter"'" a zero."
  30 LET Total=Ø
  40 INPUT "Number,please ";n
  50 LET total=Total+n
  60 PRINT "Total so far is ";To
tal
  70 IF n<>Ø THEN GO TO 4Ø
```

Fig. 4.5. A number-totalling program which can't use FOR...NEXT.

advance how many times we might want to go through the loop, so we have to go back to using GOTO. This time, however, we'll keep GOTO under closer control. We make the total variable Total equal to zero in line 3Ø. Each time you type a number, then, in response to

the request in line 4Ø, the number that you type is added to the total in line 5Ø, and line 6Ø prints the value of the total so far. Line 7Ø controls the loop, and the key to the control is the instruction word IF. IF is used to make a test, and the test in line 7Ø is to see if the value of n is not equal to zero. The odd-looking sign that is made by combining the 'less-than' and the 'greater than' signs, <>, is used to mean 'not equal', so that the line reads: 'If n is not equal to zero, then go to line 4Ø'.

The effect, then, is that if the number which you have typed in line 4Ø was not a zero, line 7Ø will send the program back to repeat line 4Ø. This will continue until you do enter a zero. When this happens, the test in line 7Ø fails (n is zero), and the program looks for a line 8Ø. Since it can't find one, it stops.

Now this allows you much more freedom than a FOR...NEXT loop, because you are not confined to a fixed number of repetitions. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now. We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 4.6. The mathematical signs are used for convenience,

Sign	Meaning
=	Equality (quantities must be identical)
<	Quantity on left is less than quantity on right
>	Quantity on left is greater than quantity on right
<=	Quantity on left is less than or equal to quantity on right
>=	Quantity on left is greater than or equal to quantity on right
<>	Quantities are not equal

Fig. 4.6. The mathematical signs that are used for comparing numbers.

and you have to remember which way round the 'greater than' and 'less than' signs have to be. It's important to note that the equals sign means 'identical to' when it is used in a test like this. If a is 5.9999999 and b is 6.ØØØØØØØ then a test such as IF a = b will fail – a is not identical to b, even though it is close enough to be equal as far as we are concerned. The important point here is that the numbers we see on the screen have been rounded, so that PRINT a in the example above might give the result 6. The test, however, is made on the numbers which have not been rounded.

Figure 4.7 shows another test – this time on string variables. The instructions are in line 2Ø; you are asked to press the y or n key. Line 3Ø gets your answer; you have to type y or n and then press ENTER.

```
  10 CLS
  20 PRINT "Please press y or n
key,then"'" ENTER"
  30 INPUT a$
  40 IF a$="y" THEN PRINT "That'
s yes": GO TO 70
  50 IF a$="n" THEN PRINT "That'
s no": GO TO 70
  60 PRINT "You cheated_try agai
n": GO TO 20
  70 PRINT "That's it!"
```

Fig. 4.7. Testing string variables, in this example to find whether a replay is y or n.

The key that you have pressed has its value assigned to a$, so that a$ should be y or n. Lines 4Ø and 5Ø then analyse this result. If both tests fail, though, the program will move from line 5Ø to line 6Ø. Your answer was not exactly y or n, so that you are asked to try again, and the GOTO 2Ø at the end of line 6Ø causes the program to repeat from line 2Ø. This line 6Ø constitutes a *mugtrap*, a way of trapping mistakes. Very often when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an incorrect entry. A good mugtrap should show the user the error of his/her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error message showing. For the skilled programmer (you, by the time you finish this book), this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop. Mugtraps are our method of ensuring this.

The test in this example is for identity. Only if a$ is absolutely identical to y will the phrase 'That's yes' be printed. If you typed a space ahead of y, or a space following, or typed Y in place of y, then a$ will not be identical, and the test fails. Failing means that a$ is not identical to y and everything that follows THEN in that line will be ignored. It's up to you to form these tests so that they behave in the way that you want.

Just to emphasise the sort of power that these simple instructions give you, Fig. 4.8 illustrates a very simple number-guessing game. Line 1Ø starts by clearing the screen, and the LET x =

```
  10 CLS : LET x=1+INT (RND*10):
 LET score=0: FOR j=1 TO 10
  20 PRINT "Guess the number. If
 you get "'"near, I'll tell you!
"
  30 PRINT "Score so far is ";sc
ore
  40 INPUT "Guess now ";n
  50 IF n=x THEN PRINT "spot on!
": GO TO 80
  60 IF ABS (n-x)<3 THEN PRINT "
Close_it was ";x: GO TO 90
  70 NEXT j: PRINT "Final score
is ";score: GO TO 9999
  80 LET score=score+2: GO TO 70
  90 LET score=score+1: GO TO 70
```

Fig. 4.8. A simple number-guessing game which uses number comparisons.

1+INT(RND*1Ø) step causes variable x to take a value that lies between 1 and 1Ø. We can't predict what this value will be, because RND means 'select at random' – a number is picked, somewhere in the range of Ø to almost 1. Because RND produces a number that is always a fraction, less than 1, multiplying RND by 1Ø will give a number that will be somewhere between Ø and 9.9999, and the INT of this will be between Ø and 9. Adding 1 gives a range of 1 to 1Ø, which is what we want. Simple, really! In line 2Ø, the instructions ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 4Ø, and the tests are made in lines 5Ø and 6Ø. If the number that you picked is identical to the random number, then you get the 'spot on!' message in line 5Ø, and the program moves to line 8Ø to add two points to your score. The less obvious test is in line 6Ø. The expression n–x is the difference between your guess, n, and the number x. If your guess is larger than the number, then n–x is a positive number. If your guess is less than x, then n–x is a negative number. The effect of ABS, however, is to make any number positive, so that if x were 5 and you guessed 6 or 4, then ABS (n–x) would come to 1. If you get a difference of 1 or 2 (less than 3), the message in line 6Ø is printed. The program then shifts to line 9Ø to present you with one point. If you don't get anywhere near, the program repeats because of its NEXT j in line 7Ø. It's very simple, but quite effective.

Reading the data

There's yet another way of getting data into a program while it is

running. This one involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item on the list. The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step.

The program in Fig. 4.9 illustrates how we can use these instructions. Line 2Ø starts a loop which will select ten items. In line

```
10 CLS : PRINT "number","root"
20 FOR n=1 TO 10
30 READ number
40 PRINT number,SQR number
50 NEXT n
60 DATA 23,14,16,56,214,99,31,
42,50,27
```

Fig. 4.9. Using the READ DATA instruction.

3Ø, the instruction 'READ number' means that a number will be selected from a list, and assigned to the variable name 'number'. The list of numbers is in line 6Ø. It is marked by the word DATA, and the computer will completely ignore this line until it comes to the READ part of the instruction. As each number is read in turn, it and its square root is printed in line 4Ø, and the next number is read. We have to be careful to match the number of items that follow DATA with the number of times we use the loop. If we try to READ eleven items, and have provided only ten, then an error message will put a stop to the program for us. We also have to match the data items with the variable names that we use for them. We can read a number item and assign it to a string variable name, but we can't read a string item and assign it to a number variable. We can, however, use several lines of DATA, and the computer will read the items in order, starting at the lowest numbered line.

The benefits of READ...DATA are not confined to numbers alone. Figure 4.10 illustrates a use of READ and DATA with strings. The aim of the program is to find out how much your shopping costs! It's based on the totalling program that we looked at earlier, with the difference that a READ takes place in line 2Ø. This reads an item from the list in line 1ØØ and tests to find if the item is called 'end'. If the item is not 'end' then line 3Ø prints its name, and calls for

```
  10 CLS : LET total=0
  20 READ a$: IF a$="end" THEN G
O TO 70
  30 PRINT "How many ";a$;" do y
ou want?";"...and at what price?
"
  40 INPUT "number ";number: INP
UT "cost ";cost
  50 LET total=total+number*cost
  60 GO TO 20
  70 PRINT "Total cost will be "
;total
 100 DATA "apples","bananas","pe
ars","peanuts","oranges","lemons
","grapefruits","end"
```

Fig. 4.10. Strings can be used in a DATA line, but remember to place quotes around each string.

the number and price to be entered in line 4Ø. The program loops round, calculating the total cost in line 5Ø each time, until the 'end' item is read. When this happens, line 7Ø gives the total cost, and the program ends. There's one more twist to the use of the READ and DATA instructions, in the form of RESTORE, but we'll reserve that one for the next chapter.

Single key reply

So far, we have been putting in y or n replies with the use of INPUT, which means pressing the key and then pressing ENTER. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press ENTER. For snappier replies, however, there is an alternative in the form of INKEY$. INKEY$ is an instruction that carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by placing the INKEY$ instruction in the loop which will repeat until a key is pressed. Figure 4.11 shows such a loop. The INKEY$

```
  10 CLS
  20 PRINT "Press any key to pro
ceed"
  30 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 30
  40 PRINT "the key was ";k$
  50 REM some keys do not give a
 character on the screen. The EN
TER key has been excluded.
```

Fig. 4.11. Using INKEY$ for a single-key reply.

instruction will produce a string quantity when any key is pressed, so we assign INKEY$ to a string variable, k$. In this way, when any key is pressed, the quantity that it represents will be assigned to k$, and if k$ is a 'blank string', meaning that no key was pressed, the line loops back to its start again. Note how we indicate a blank string by using two quotes with no space between them. We also have to reject the action of the ENTER key itself, however, and this is done by the second test, comparing k$ with CHR$ 13. Failing to do this can make the action erratic – I'll explain the use of CHR$ later, so trust me!

Subroutines and menus

A *subroutine* is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is inserted by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an automatic return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point that immediately follows the GOSUB. Figure 4.12 illustrates this. When the program

```
  10 CLS : PRINT "This is a ";
  20 GO SUB 1000
  30 PRINT "subroutine"''
  40 PRINT "Red light and green
light make": GO SUB 1000: PRINT
" light."
  50 PRINT "wasps have ";: GO SU
B 1000: PRINT "and black stripes
."
  60 GO TO 9999
1000 PRINT "yellow ";
1010 RETURN
```

Fig. 4.12. Using a subroutine – this is the key to more advanced programming.

runs, line 10 prints a phrase, with the semicolon used to prevent a new line from being selected. The GOSUB 1000 in line 20 then causes the word 'yellow' to be printed, but the RETURN in line 1010 will send the program back to line 30, the instruction that immediately follows the first GOSUB 1000. This action will also occur even when the GOSUB is part of a multistatement line, as lines 40 and 50 demonstrate. The GOSUB 1000 will cause the word yellow to be printed, but the return is to the PRINT instruction that follows

GOSUB 1ØØØ in line 5Ø, it doesn't jump to line 6Ø. This example is, of course, a yellow subroutine. One more point is important. We should not allow the subroutine to run except when it has been requested by the use of GOSUB. This is done by putting GOTO 9999 at the end of the main section of the program, ahead of any subroutines that we use. GOTO 9999 is the Spectrum equivalent of an 'end' instruction, because the Spectrum does not allow you to use any line number greater than this.

Now to see one of the most useful applications of subroutines, we have to hark back to choices. A choice of two items, such as in Fig. 4.7, isn't exactly generous. We can extend the choice by a program routine that is called a *menu*. A menu is a list of choices, usually of program actions. By picking one of these choices,we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. We could use a set of lines such as:

IF K =1 THEN 1ØØØ
IF K = 2 THEN 2ØØØ

and so on. Figure 4.13 shows an improvement on this one, making use of subroutines. It's a feature of Spectrum that we can use an

```
  10 CLS : PRINT AT 1,14;"MENU"
  20 PRINT ''
  30 PRINT "1.Enter program name
."
  40 PRINT "2.enter amount of RA
M used."
  50 PRINT "3.List all programs.
"
  60 PRINT "4.List programs belo
w a "" specified size."
  70 PRINT '"Please select by nu
mber."
  80 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 80
  90 LET k=VAL k$: IF k>4 OR k<1
 THEN PRINT "Incorrect selection
,please try again.": GO TO 70
 100 GO SUB k*1000
 110 GO TO 9999
1000 PRINT "name section": RETUR
N
2000 PRINT "RAM section": RETURN

3000 PRINT "List all": RETURN
4000 PRINT "Select list": RETURN
```

Fig. 4.13. A menu choice that makes use of subroutines.

expression, like 'a+1Ø' or '25*a−1Ø' as the line number following a GOSUB. The menu prints the choices and allocates each one to a number. The INKEY$ routine in line 8Ø waits for you to press a key – but the value of k$ that it obtains is a string value. This string value is converted into a number value in line 9Ø, but using VAL. VAL applied to any string will extract a number value, if there is one contained in the string. This number is allocated to variable k, and used to select the GOSUB lines in line 1ØØ. I haven't written out the GOSUB lines in full, because it's just too much typing for the sake of a demonstration. The PRINT lines, however, prove that the subroutine action has been carried out. Notice how the value of k is tested in line 9Ø – it's another example of mugtrapping in use.

Flasher's joy

The trouble with INKEY$ is that it doesn't remind you that it's in use – there's no question mark printed as there is when you use INPUT. The program of Fig. 4.14 shows a useful variation on INKEY$ for use in menus. The subroutine in lines 1ØØØ to 1Ø4Ø causes an asterisk to flash while you are thinking about which key to press. The asterisk is

```
  10 CLS
  20 PRINT "Choose 1 or 2 please
"
  30 GO SUB 1000
  40 LET a=VAL k$
  50 PRINT "Choose y or n please
": PAUSE 10: LET k$=""
  60 GO SUB 1000
  70 LET b$=k$
  80 PRINT '''"You chose ";a;" an
d ";b$
  90 GO TO 9999
1000 LET k$=INKEY$
1010 IF k$<>"" AND k$<>CHR$ 13 T
HEN RETURN
1020 PRINT AT 5,1;"*": PAUSE 10
1030 PRINT AT 5,1;" ": PAUSE 10
1040 GO TO 1000
```

Fig. 4.14. Using an INKEY$ input along with a flashing asterisk.

flashed by alternately printing the asterisk and the delete step. Meantime, make friends with subroutines. They are not just a useful way of obtaining an action at several points in a program, they are an indispensable aid to program planning, of which there's much more in Chapter 6.

Chapter 5
String Along With Spectrum

In Chapter 3, we took a fairly brief look at number functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's look at an example. Figure 5.1 shows a program that prints 'Spectrum tricks' as a title. What makes it more eye-

```
  10 CLS
  20 LET t$="Spectrum tricks"
  30 GO SUB 1000
  40 PRINT
  50 LET t$="▚▚▚▚▚▚▚▚▚▚▚▚▚▚▚▚"
  60 GO SUB 1000
  70 GO TO 9999
1000 PRINT TAB (16-LEN t$/2);t$
1010 RETURN
```

Fig. 5.1. Introducing LEN, a member of the string function family.

catching is the fact that the words are printed with an underlining of chequer shapes. This is your first introduction to the shapes, called *graphics*, that the Spectrum can print in addition to numbers and letters. We'll go into more detail on these shapes in Chapter 7 but, for the moment, we'll just use the shapes without much explanation. In line 5ϕ, after putting in the line as far as the first quotes, you have to 'enter graphics mode'. This is done by pressing the CAPS SHIFT and the '9' keys together. You will see the cursor letter change to a 'G', and any number key that you use now will produce a shape – the pattern that is printed on it. The chequer shape that we want is on the '6' key, so press this key fifteen times. Then press CAPS SHIFT and '9' again to

leave graphics mode, and add the final quote mark in the usual way.

Now what this program does is to make use of a subroutine to print 'Spectrum tricks' and its underlining of chequers centred along each line. The centring is carried out in line 1ØØØ, and this is the line that introduces the first string function of this chapter. LEN is used to find how many characters (letters, digits, punctuation marks or spaces) exist in a string variable. LEN t$ will come up with the number of characters in the variable t$, whatever t$ may be. What we're doing in line 1ØØØ, then, is to get the computer to carry out the actions that we looked at in Chapter 2, for centring a phrase. LEN t$/2 will give the number that is half the number of characters, and this is then subtracted from 16 to give the correct TAB number. The TAB instruction ignores fractions, so it doesn't matter if t$ contains an odd number of characters. Notice, by the way, that if we want anything printed centred by this subroutine, we have to give it the variable name of t$. This action is called 'passing a variable' to the subroutine, and it's something that we have to keep a careful eye on when we use subroutines. You can't expect a subroutine that is written to print t$ centred to have any effect on a string called a$.

A slice in time

The next group of string operations that we're going to look at are called *slicing operations*. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string. All of that might not sound terribly interesting, so take a look at Fig. 5.2. The string a$ is assigned in line 1Ø, and sliced in line

```
  10 CLS : LET a$="Speciaℓ ten-d
olℓar rum"
  20 LET b$=a$( TO 4)+a$(9)+a$(2
0 TO )
  30 PRINT b$
```

Fig. 5.2. Spectrum string slicing.

2Ø. What's printed in line 3Ø is the word Spectrum. Now how did this happen? The instruction TO is used on the Spectrum, and means 'copy part of a string'. As normally used, TO has to have a number ahead of it, and another following it. The first of these numbers is the position for starting the slice. This position is counted from the left-hand side. The second number is the position at which the slice stops,

again counted from the left-hand side. If the first number is omitted, we start slicing at the left-hand side. If the second number is omitted, we stop slicing at the right-hand side. If that sounds complicated, then let's take a look at line 2Ø. The first slice is a$(TO 4). We need to specify what string we want to slice, so a$ has to appear. Starting with TO means that we want to start the slice at the left-hand side of the string, and the '4' following the TO means that we stop at the fourth character. The fourth character of a$ is the 'c' of 'Special', so that the effect of a$(TO 4) is to copy 'Spec'. The + sign then *concatenates* – remember! – to the next string. This next one is simply a$(9), with no TO. It simply means the ninth character, counting from the left, which is 't'. Don't forget that the space between 'Special' and 'ten-dollar' counts as a character. Finally, we slice a$(2Ø TO), which means that we take from character number 2Ø to the end. That gives us 'rum', and tacking this on to what we have already produces 'Spectrum'. It's a long road for a short-cut, but it illustrates how simple this type of string-slicing action is. You will find this method only on the ZX computers. You may find, in programs that have been written for other computers, instructions called LEFT$, RIGHT$ and MID$. Figure 5.3 shows how these LEFT$, RIGHT$ and MID$ instructions can be converted into the TO instructions. One of the

LEFT$(A$,2)	Slice A$ starting at the second letter. Spectrum equivalent is A$(2 TO).
RIGHT$(A$,4)	Slice A$ for four characters starting from the right-hand side. Spectrum equivalent is A$(LEN A$–4 TO).
MID$(A$,2,5)	Slice A$ starting at the second character, and taking 5 characters. Spectrum equivalent is A$(2 TO 6).

Fig. 5.3. Converting the string slicing instructions of other computers to the Spectrum form.

features of all of these string slicing instructions, incidentally, is that we can use variable names or expressions in place of numbers.

Code and CHR$

Some of the most useful string operations make use of ASCII code numbers. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced 'Askey') code

is one that is used by most computers. Figure 5.4 shows a printout of the ASCII code numbers and the characters that they produce on my ZX printer. The number characters of standard ASCII code extend only from 32 to 127. We can find out the code for any letter by using

```
32        56  8       80  P       104 h
33  !     57  9       81  Q       105 i
34  "     58  :       82  R       106 j
35  #     59  ;       83  S       107 k
36  $     60  <       84  T       108 l
37  %     61  =       85  U       109 m
38  &     62  >       86  V       110 n
39  '     63  ?       87  W       111 o
40  (     64  @       88  X       112 p
41  )     65  A       89  Y       113 q
42  *     66  B       90  Z       114 r
43  +     67  C       91  [       115 s
44  ,     68  D       92  \       116 t
45  -     69  E       93  ]       117 u
46  .     70  F       94  ↑       118 v
47  /     71  G       95  _       119 w
48  0     72  H       96  £       120 x
49  1     73  I       97  a       121 y
50  2     74  J       98  b       122 z
51  3     75  K       99  c       123 {
52  4     76  L       100 d       124 |
53  5     77  M       101 e       125 }
54  6     78  N       102 f       126 ~
55  7     79  O       103 g       127 ©
```

Fig. 5.4. The ASCII code numbers that are used to represent string characters.

the function CODE, which is followed by a string character. The result of CODE is a number, the ASCII code number for that character. If you use CODE a$, where a$ is a collection of characters, then you'll get the code for the first character only, because the action of CODE includes rejecting more than one character. Figure 5.5 shows CODE in action. String variable a$ is assigned in line 1Ø and in

```
10 CLS : LET a$="Spectrum"
20 FOR j=1 TO LEN a$
30 PRINT CODE a$(j);" ";
40 NEXT j
```

Fig. 5.5. Using CODE to find an ASCII code number. Other computers use ASC for this purpose.

line 2Ø a loop starts which will run through all the letters in a$. The letters are picked out one by one, using the slicing action which doesn't need TO, and the ASCII code for each letter is found with CODE. The space between quotes, along with the semicolons in line

3Ø makes sure that the codes are all printed on one line with a space between the numbers. Simple, really.

CODE has an opposite function, CHR$. What follows CHR$ has to be a code number, and the result of the action is the character whose code number is given. The instruction PRINT CHR$ 65, for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this, just to give one example, for hiding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.6 illustrates this use. Line 4Ø contains an INKEY$ loop to make the program wait for you. When

```
  10 CLS : PRINT ''
  20 PRINT "White light into a p
rism will";"produce _ what?"
  30 PRINT "Press any key for an
 answer."
  40 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 40
  50 FOR j=1 TO 10: READ d
  60 PRINT CHR$ d;: NEXT j
  70 DATA 65,32,83,112,101,99,11
6,114,117,109
```

Fig. 5.6. Using ASCII codes to conceal a message.

you press a key, the loop that starts in line 5Ø prints ten characters on the screen. Each of these is read as an ASCII code from a list, using a READ instruction in the loop. The PRINT CHR$ d in line 6Ø then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 2Ø, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent de-coders!

Order, order!

We saw earlier, in Fig. 4.8, how numbers can be compared and we have also looked at testing strings for equality. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not

difficult to see what the identity sign, =, means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.7 illustrates this use of comparison using the = and > symbols. Line 2Ø assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 3Ø

```
  10 CLS
  20 LET a$="qwerty"
  30 PRINT : INPUT "type a word
";b$
  40 IF b$=a$ THEN PRINT "Same a
s mine!": GO TO 9999
  50 IF a$>b$ THEN LET c$=a$: LE
T a$=b$: LET b$=c$
  60 PRINT "Order is ";a$;" then
 ";b$
```

Fig. 5.7. Comparing words to decide on their alphabetical order.

then asks you to type a word. The comparisons are then carried out in lines 4Ø and 5Ø. If the word that you have typed, which is assigned to b$, is identical to qwerty, then the message in line 4Ø is printed, and the program ends. If 'qwerty' comes earlier in an index than your word, then line 5Ø is carried out. If, for example, you typed 'tape', then since t comes after q in the alphabet and has an ASCII code that is greater than the code for q, your word b$ scores higher than a$, and line 5Ø has no effect. If the word that you typed 'comes earlier' than qwerty, meaning that its ASCII codes are lower, then a$ is 'greater than' b$. Suppose, for example, that you type 'peripheral' as your word. This is 'less than' qwerty and the second part of line 5Ø swaps them round. This is done by assigning a new string, c$ to a$ (so that c$ = "qwerty"), then assigning a$ to b$ (so a$ = "peripheral"), then b$ to c$ (so that b$="qwerty"). Line 6Ø will then print the words in the order a$ and then b$, which will be the correct alphabetical order, and the test in line 5Ø fails. Note the important point, though, that words like qwertz and qwertx will be put correctly into order – it's not just the first letter that counts.

Numbers in array

The variable names that we have used so far are useful, but there's a limit to their usefulness. Figure 5.8 illustrates this. Lines 1Ø to 4Ø

```
10 CLS : DIM a(10)
20 FOR n=1 TO 10
30 LET a(n)=5+INT (RND*90)
40 NEXT n: PRINT
50 PRINT TAB 11;"Marks list"
60 PRINT : FOR j=1 TO 10
70 PRINT "Item ";j;" received
";a(j);" percent."
80 NEXT j
```

Fig. 5.8. An array of subscripted variables. It's simpler than the name suggests!

generate an (imaginary) set of examination marks. This is done using random numbers simply to avoid the hard work of entering the real thing. No, it's not the way that the examiners arrive at their marks! The variable in line 3Ø is something new, though. It's called a subscripted variable, and the 'subscript' is the number that is represented by n. The name that we use has nothing to do with computing – it's a name that was used long before computers were around. How often do you make a list with the items numbered 1,2,3 ... and so on? These numbers 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names a(1), a(2), a(3) and so on, we can identify different items that have the common variable name of a. The whole group is called an *array*. A member of this group like a(2) has its name pronounced as 'a-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number variable or an expression, this allows us to work with or pick out any item of the list. Figure 5.8 shows the list being constructed from the FOR...NEXT loop in lines 2Ø to 4Ø. Each item is obtained by finding a random number between 5 and 95, and is then assigned to a(n). Ten of these 'marks' are assigned in this way, and then lines 5Ø to 8Ø print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

You can't just rush into the subscripted variable business, though. The computer has to keep track of all these different values, and this needs a certain amount of organisation. This means instructing the computer to prepare space. This is done by using the instruction word

DIM. DIM means 'dimension', and the instruction consists of naming each variable that you will use for arrays, and following the name with the maximum number, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you attempt to use a number that is higher than the one you have put into the DIM instruction, then the computer will stop with an error message – '3 Subscript wrong'. You will have to change the DIM instruction and start again – which will be tough luck if you were typing in a list of a hundred names! Note that you will have to dimension for every array that you use.

```
 10 CLS : DIM a$(10,16)
 20 PRINT "Please enter surname
s."
 30 FOR j=1 TO 10
 40 PRINT "Name ";j: INPUT a$(j
)
 50 NEXT j
 60 PRINT "All done"
 70 PAUSE 100
 80 CLS : FOR n=1 TO 10 STEP 2
 90 FOR s=0 TO 1
100 PRINT a$(n+s),: NEXT s
110 NEXT n
```

Fig. 5.9. Using subscripted string variables. You have to dimension this type of array carefully.

Figure 5.9 extends this another step further. This time you are invited to type a name for each of ten items. After you have pressed ENTER, the name that you have typed is assigned as one of the items of the array, a$. When the list is complete, there is a pause so that you can get your breath back. The pause is programmed, logically enough, by using the instruction PAUSE. PAUSE has to be followed by a number which specifies how long the pause will be. The pauses are measured in fiftieths of a second (sixtieths in the USA and Japan), so that PAUSE 50 will give a one-second pause in Britain but PAUSE 60 is needed in USA and Japan. The pause that we've used is PAUSE 100, which gives a two-second delay. The screen is then cleared. The list is then printed neatly, using line 100. This illustrates another nested loop. The loop that starts in line 90 uses the variable 's'. When s is zero, value of n is 1, for example, then what is printed is a$(n+s), which is a$(1). The comma then forces printing to the centre of the screen, and the NEXT s takes effect. This causes the value of n+s to become 2, so that a$(2) is printed. That's the end of the s loop this time round, so line 110 is carried out, taking a new line. It's a very

simple way of printing the names in two columns. The important point about this program is that it demonstrates that it's not just numbers that we can keep in this array form. One point you have to watch, though, is that the name of a string array must not be the same as the name of an ordinary string variable. You can have a number variable n and a number array n( ) existing together, but a string array a$( ) will delete a string called a$. Unless you use a lot of variables in a program, it's not exactly a handicap, but you do need to be careful about how you select names for string arrays.

There is one novelty that I have dodged so far, though. Line 1Ø of Fig. 5.9 shows a new twist to the use of DIM. As we used it previously, DIM had to be used when we had a number array, and the number that followed DIM was the number of items in the array. We also have to dimension a string array, and this is rather more complicated than dimensioning a number array. A Spectrum string array needs two dimensioning numbers within the brackets. One of these is the number of items in the array, as before. The other is the maximum number of characters in a string. If, for example, you dimension to have 20 characters, then you can have any number up to twenty, but no more. You have to decide in advance what will be a reasonable number of characters per string item. If you choose a large number, like 100, then you won't be able to use many string items. That's because each of them will need 100 units, called *bytes*, of memory. This amount will be needed whether your strings are 100 characters long or not – it's what you have dimensioned that counts. 100 string items at 100 characters each is ten thousand bytes of memory used! If your string items are shorter than the maximum that you have set, Spectrum will automatically fill them out with spaces. If they are too long, letters will be chopped off from the end!

As a matter of record ...

Entering items into arrays is hard work, particularly if you are a one-finger typist. It's certainly an activity that you don't want to repeat if you can possibly avoid it. Spectrum allows you to avoid it by saving (which means recording) arrays on cassette. The ZX Microdrive will be available later for this task, also.

The program in Fig. 5.10 illustrates how an array, a number array in this example, is saved. The steps up to line 4Ø should be familiar territory to you by now, so we'll concentrate on 4Ø onwards. Line 4Ø

```
  10 CLS : DIM a(50)
  20 FOR j=1 TO 50: LET a(j)=15+
INT (RND*75)
  30 NEXT j
  40 PRINT "Now prepare to recor
d the list."'"Make sure that you
 have a "'"cassette in the recor
der."
  50 SAVE "marks" DATA a()
  60 CLS : PRINT "All done_now v
erify it."'"Please rewind the ca
ssette."
  70 PAUSE 250: PRINT "Now play
back"
  80 VERIFY "marks" DATA a()
  90 PRINT "All done."
```

Fig. 5.10. Saving a number array on a cassette. Other data, including pictures on the screen can also be saved (see your Spectrum Manual for details).

delivers a message to tell you what's happening. This is essential, because you can't make a recording until you have a cassette in the recorder – with no reminder like this, you could easily forget. You also have to remember to pull out the EAR plug from the recorder, if you haven't already got into the habit. Line 50 is the data-saving line. The keyword is SAVE, and it has to be followed by a *filename*. The filename is important, because the Spectrum uses it to recognise the recording when you come to play it back. By using different filenames, you can have several data recordings on a tape and allow the computer to select the one that you want.

The next item following SAVE is the word DATA. This prepares the computer for saving data as distinct from pictures on the screen. The last item is the variable name of the array that you want to save, followed by a pair of brackets. You don't need to put anything, not even a space, between the brackets. Note that the filename is enclosed in quotes, but the array name isn't. When the Spectrum comes to this line in the program, it will deliver the message, 'Start tape, then press any key'. You must then press the PLAY and RECORD keys of your recorder, and then press any of the Spectrum keys. The data array will then be recorded.

Lines 60 to 80 then verify that the recording was O.K. After all, if it took you an hour to type all the data in, you don't want to take any risks. A bad piece of tape, a bit of electrical interference while you were recording – these things could cause a bad recording. The VERIFY procedure follows familiar lines, with the same items following VERIFY as followed SAVE.

The playback steps (loading data) follow the same pattern. I've illustrated them in Fig. 5.11. This time, I've used an INKEY$ step to keep the computer waiting in line 40. This gives you time to prepare the cassette, put the EAR plug in place, and rewind the tape to the

```
  10 CLS : DIM m(52)
  20 PRINT TAB 11;"Marks List"
  30 PRINT "Please prepare data
cassette"'"to replay data. Press
 any key"'"when ready."
  40 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 40
  50 PRINT "Loading now": LOAD "
marks" DATA m()
  60 FOR j=1 TO 49 STEP 4: FOR s
=0 TO 3: IF j+s>50 THEN GO TO 99
99
  70 PRINT TAB 7*s;m(j+s);
  80 NEXT s: PRINT : NEXT j
```

Fig. 5.11. The steps that are needed when an array is replayed from a cassette.

correct place. The first part of line 50 delivers the message – it's important to know what is going on. The second part of line 50 then loads the array, using the instruction word LOAD. Once again, this has to be followed by the filename, DATA, and the array name, then the pair of brackets. If you start the tape running by pressing the PLAY key when you see the 'Loading now' message, then you will soon be rewarded by seeing the data appear. Lines 70 and 80 carry out the printing action. If you didn't want to print the data right away, your program could contain another PRINT line following the LOAD. A message such as:

Data loaded – please stop recorder

is very useful. Note, incidentally, how the marks are printed in four columns by using the TAB number 7*s.

Chapter 6
Do It Yourself!

You can get a lot of enjoyment from your Spectrum when you use it to enter programs from cassettes that you have bought. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of GWR steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move across the screen. Programs of that type will need instructions that we have not looked at yet, and they are dealt with in the next three chapters. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all types of programs.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an

elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. For myself, I use a 'student's pad' of A4 which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, I said sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a fairly simple program, but it will illustrate all the skills that you need. The two programs that appear, with comments, at the end of this book will give you further experience in exploring design for yourself.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it!

As an example, take a look at Fig. 6.1. This shows a program outline plan for a simple game. The aim of the game is to become familiar with units of money around the world. The program plan shows what I expect of this game. It must present the name of a country on the screen, and then ask what the main unit of money is (like pound, dollar, mark and so on). A little bit more thought

Aims:

- Present the name of a country on the screen
- Ask what the main unit of currency is
- Must be correctly spelled
- Must avoid user being able to read answer from listing
- One point for each correct answer
- One additional try allowed
- Keep track of attempts
- Present score as number of successes out of number of attempts
- Pick country names at random

Fig. 6.1. A program outline plan. This is your starter!

produces some additional points. The name of the currency will have to be correctly spelled. A little bit of trickery will be needed to prevent the user (son, daughter, brother, sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program the way an artist paints a picture. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 6.2 shows what this might look like at this stage.

The BASIC foundations

Now at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages

- Display title, then instructions
- Present name of country on screen
- Ask for unit of currency
- Input for reply
- Compare with correct answer
- If correct, ask if another one is wanted
- If incorrect, give one more try
- If second try is incorrect, select another country
- Ends when user types n in reply to 'Do you want another one?'

Fig. 6.2. The next stage in expanding the outline.

with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 6.3 shows what you should aim for at this stage. There are only nine lines of program here, and that's as much as you want. This is a foundation, remember, not the Albert Hall. It's also a program that is being developed, so we've hung some 'danger – men at work' signs around. These take the form of the lines that start with REM.

```
10 CLS : GO SUB 1000
11 REM title
20 GO SUB 1200
21 REM Instructions
30 GO SUB 1400
31 REM Dimensions and arrays
40 GO SUB 2000
41 REM Play
50 GO SUB 3000
51 REM Score
60 GO SUB 4000
61 REM Another?
70 IF k$="y" OR k$="Y" THEN GO
TO 40
80 GO TO 9999
90 REM END
```

Fig. 6.3. A 'core' or 'foundation' program for the example.

REM means REMinder, and any line of a program that starts with REM will be ignored by the computer. This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In Fig. 6.3, I have put the REM notes on lines

which are numbered just 1 more than the main lines. This way, I can remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they make a program take up more space in memory, and run slightly slower. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for everyday use, and a fully-detailed version that I can use if I want to make changes.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 6.2 exactly, and the only part that is not committed to a GOSUB is the IF in line 7Ø. What we shall do is to write a subroutine which will use INKEY$ to look for a 'y' or 'n' being pressed, and line 7Ø deals with the answer. What's the question? Why, it's the 'Do you want another game' step that we planned for earlier.

Take a good long look at this piece of program, because it's important. The use of all the subroutines means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
9 GO TO 3Ø
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it.

The next step is to get to the keyboard (at last, at last!) and enter this core program. If you use the GO TO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of Fig. 4.13, so you know how to go about it. This allows

you to test your core program and be sure that it will work before you go any further.

There are two ways of going further. One is to record each piece of the program on tape as you go along. The Spectrum has a MERGE command, which allows you to construct a long program from short sections by adding pieces from tape. The alternative, which I prefer, is to keep adding to the core. If you have the core recorded, then you can load this into your Spectrum, add one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another cassette. Next time you want to add a subroutine, you start with this version, and so on. This way, you keep tapes of a steadily growing program, with each stage tested and known to work.

Subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine that is to be placed in line 4000. This is just our familiar INKEY$ routine, along with a bit of PRINT, so we can deal with it right away. We may even have this piece of routine recorded on tape (all neatly labelled, with a note of the line numbers, I hope). If we do, then we can use MERGE to run this on to our core program. The editing procedures of the Spectrum will then allow us to renumber the lines of the recorded subroutine to the numbers of 4000 onwards that we want.

```
4000 PRINT "Would you like anoth
er one?"'"Please press y or n ke
y."
4010 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 4010
4020 RETURN
```

Fig. 6.4. The subroutine for line 4000.

If we don't have it on record, we will have to write it, and Fig. 6.4 shows the form it might take. The subroutine is straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this subroutine in place.

The hard part

Now we come to what you might think is the hardest part of the job –

the subroutine which carries out the 'Play' action. In fact, you don't have to learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a subroutine to be dealt with later.

As an example, take a look at Fig. 6.5. This is a plan for the Play subroutine, which also includes information that we shall need for the

---

- Keep answers as a set of ASCII codes
- Each DATA line of codes ends with Ø
- Keep list of countries in an array, q$
- The number of the array item can be used to select the DATA line for the answer
- Use variable 'try' to record tries, and 'score' to record number of successes
- Use 'go' to keep track of the number of attempts at one question (limit to 2)

Fig. 6.5. Planning the 'Play' subroutine.

setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of ASCII codes, with a Ø as the last number. Why Ø? The answer is as a 'terminator' (remember?), so that the computer reads the correct number of codes each time. That's the first item for this subroutine.

The next one is that we shall keep the names of the countries in an array. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 1Ø, then we can place the answers in DATA lines whose numbers are, for example, 6ØØ1 to 6Ø1Ø. We can put the DATA line number in the form 6ØØØ + n, where n is the number of the item. What I'm aiming at is the use of an instruction 'RESTORE number'. This will make the next READ use the line whose number follows the RESTORE instructions. If, for example, we use RESTORE 6ØØ5, then the next READ instruction will read the data starting at line 6ØØ5, and so on.

RESTORE 6ØØØ+n looks like being useful to us for locating the answers.

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using 'score' for the score and 'try' for the number of tries looks self-explanatory. The third one, 'go' is one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the array that will hold the country names – q$().

Play for today

Figure 6.6 shows what I've ended up with as a result of the plan in Fig. 6.5. The steps are to pick a random number, use it to print a country name, and then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another subroutine.

```
2000 LET go=0: RANDOMIZE : LET v
=1+INT (RND*10)
2001 REM Picks v at random betwe
en 1 and 10
2010 CLS : PRINT AT 5,3;"The cou
ntry is ";q$(v)
2020 PRINT AT 7,3;"The currency
is the_ ";
2030 INPUT "Please answer here-"
;x$: PRINT x$: LET try=try+1
2040 GO SUB 5000
2041 REM Find correct answer,a$
2050 RETURN
```

Fig. 6.6. The program lines for the 'Play' subroutine.

Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

We start in line 2ØØØ with a new instruction, RANDOMIZE. This doesn't do anything visible, but it's useful all the same. What it does is to 'reshuffle' the random number generator of the Spectrum, so that there is no chance of a set of numbers repeating. In a simple game like this, it's not much of a risk, but it's useful to know for the future. The second part of line 2ØØØ then picks a number, at random, lying between 1 and 1Ø. As before, we use line 2ØØ1 to hold a REM that reminds us of what's going on. Lines 2Ø1Ø to 2Ø3Ø are straightforward stuff. We print the name of the country that corresponds to the

random number, and ask for an answer, the currency of that country. The last section of line 2Ø3Ø counts the number of attempts. This is the logical place to put this step, because we want to make the count each time there is an answer. Now it's chicken-out time. I don't want to get involved in the reading of ASCII codes right now, so I'll leave it to a subroutine, starting in line 5ØØØ, which I'll write later. The REM in line 2Ø41 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN.

Down among the details

With the Play subroutine safely on tape, we can think now about the details. The first one to look at should be one that precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and Fig. 6.7 shows the plan. Each time that there is a correct answer, the number variable 'score' will be incremented, and we can go back to the main program. More is needed if the answer does not match exactly. We need to print a message, and allow another go. If the result of this next go is not correct, that's an end to the attempts.

Figure 6.8 shows the program subroutine developed from this

- For correct answer, increment 'score'
- For first incorrect answer (when go=Ø), try again, make go=1
- For second incorrect answer (when go=1), pass to next question
- Make go=Ø before next question is asked

Fig. 6.7. Planning the 'Score' subroutine.

```
3000 PRINT : IF x$=a$ THEN LET s
core=score+1: PRINT "Correct_you
r score is now ";score;" in ";tr
y;" attempts.": PAUSE 200: GO TO
 3030
3001 REM Correct answer
3010 IF go=0 THEN PRINT "Not cor
rect_but it might be "'" your sp
elling! You get another try free
": LET go=1: PAUSE 150: GO SUB 2
010: GO TO 3000
3011 REM First incorrect
3020 LET go=0: PRINT "No luck_tr
y the next one.": PAUSE 150
3021 REM Second incorrect
3030 RETURN
```

Fig. 6.8. The 'Score' subroutine written.

plan. Line 3ØØØ deals with a correct answer. The GO TO 3Ø3Ø ensures that if the answer was correct, the rest of the subroutine is skipped, and the subroutine returns. If the answer is not correct, though, line 3Ø1Ø swings into action. This prints a message, and then calls the subroutine at line 2Ø1Ø again so that the user can make another answer entry. The GO TO 3ØØØ at the end of line 3Ø1Ø then tests this answer again.

Now there's a piece of cunning here. The number variable 'go' must start with a value of Ø (make a note of it!). When there is an incorrect answer, however, and 'go' is still Ø, line 3Ø1Ø is carried out. One of the actions of line 3Ø1Ø, however, is to set 'go' to 1. When you answer again, with go=1, line 3ØØØ will be used, and if your second answer is wrong, line 3Ø1Ø cannot be used, because 'go' is not zero. The next line that is tried, then, is 3Ø2Ø. This puts 'go' back to zero for the next round, prints a sympathetic message, pauses, and then lets the subroutine return in line 3Ø3Ø.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 6.9 shows the subroutine that deals with dimensioning and arrays. Line 14ØØ sets all the variables for the scoring system to zero. Line 141Ø dimensions the array q$ that will be used for the names of the countries. Line 142Ø then reads the names from a data list into the array – and that's it! We can write the DATA lines later, as usual.

```
1400 LET try=0: LET score=0: LET
 go=0
1401 REM All variables to zero
1410 DIM q$(10,12)
1411 REM Countries list
1420 FOR j=1 TO 10: READ q$(j):
NEXT j
1421 REM read list
1430 RETURN
```

Fig. 6.9. The dimensioning and array subroutine.

Next comes the business of finding the answer. We have planned this, so it shouldn't need too much hassle. Figure 6.10 shows the

```
5000 RESTORE 6000+v: LET a$=""
5001 REM Find correct data line
5010 READ n: IF n=0 THEN GO TO 5
030
5020 LET a$=a$+CHR$ n: GO TO 501
0
5030 RETURN
```

Fig. 6.10. Checking the answer.

program lines. The variable v is the one that we have selected at random, and it's used to select a DATA line. Lines 5Ø1Ø to 5Ø2Ø read numbers from the DATA line, stopping when a Ø is found. Note that we have to check for a Ø before we convert the number to a letter and add it to the string. Line 5Ø2Ø builds up the answer string, which we call a$. This is set to a blank in the last part of line 5ØØØ to ensure that

```
1200 CLS : PRINT TAB 10;"Instruc
tions"'
1210 PRINT TAB 2;"You will be pr
ovided with the";"name of a coun
try. You should";"then type the
name of its main";"unit of money
,then press ENTER.";"Correct spe
lling is essential!";"The comput
er will keep score."
1220 PAUSE 350
1230 RETURN
```

Fig. 6.11. The instructions – always leave these until you have almost finished.

we always start with a blank string, not with the previous answer. That's the hard work over. Figure 6.11 is the subroutine for the instructions, and Fig. 6.12 is the title subroutine. Each of them includes a pause, and the title has another trick to it, of which much more later. Finally, Fig. 6.13 shows the DATA lines.

```
1000 PRINT TAB 9; FLASH 1;"THE M
ONEY GAME"
1010 PAUSE 250
1020 RETURN
```

Fig. 6.12. The title subroutine.

```
5500 DATA "France","Austria","Co
lombia","Denmark","India","Italy
","Holland","Portugal","Venezuel
a","Yugoslavia"
6001 DATA 70,114,97,110,99,0
6002 DATA 83,99,104,105,108,108,
105,110,103,0
6003 DATA 80,101,115,111,0
6004 DATA 75,114,111,110,101,0
6005 DATA 82,117,112,101,101,0
6006 DATA 76,105,114,101,0
6007 DATA 71,117,105,108,100,101
,114,0
6008 DATA 69,115,99,117,100,111,
0
6009 DATA 66,111,108,105,118,97,
114,0
6010 DATA 68,105,110,97,114,0
```

Fig. 6.13. The DATA lines that are needed.

Now we can put it all together, and try it out. Because it's been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM in line 141Ø. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. Take this as a sort of BASIC 'Meccano set' to reconstruct any way you like. It will give you some idea of the sense of achievement that you can get from mastering your Spectrum!

Chapter 7
Special Effects

Any modern computer is expected to be able to produce dazzling displays of colour and other special effects. The Spectrum is no exception, and in this Chapter, we'll start to look at some of the effects that are possible. To start with, we have to know some of the terms that are used, and the first of these is *graphics*. Graphics means pictures that can be drawn on the screen, and all modern computers have instructions that allow you to draw such patterns. In connection with these patterns, you'll see the words *low resolution* and *high resolution* used. 'Resolution' isn't such an easy term to explain. Imagine that you are creating pictures on a sheet of paper about eleven inches across by eight inches deep – that's roughly the size of a TV screen that is described as being a 14 inch screen (it's about 14 inches diagonally!).

Now if you are asked to create the pictures by using rectangles of coloured paper, you are dealing with picture making in a way that is very similar to the way that the computer operates. Suppose that you are allowed only 704 pieces of paper, of such a size that all 704 will fill the screen. You couldn't draw very finely detailed pictures with so few large pieces, and this is what we mean by *low resolution*. On the other hand, if you were provided with pieces so small that you would need 45056 of them to fill an entire screen size, you could produce very much more detailed pictures. This is what we mean by *high resolution*. The Spectrum has both low and high resolution graphics available, and the figures that I have used correspond to the size of the blocks that the Spectrum uses.

Vivid impressions

The best place to start on our exploration of special effects is with the

PRINT modifiers. As the name suggests, these cause changes in the appearance of anything that is printed on the screen. This action is the same no matter what we print – letters, digits, or graphics shapes.

```
10 CLS : PRINT "Normal print"
20 INVERSE 1
30 PRINT "Inverse on"
40 PRINT "Effect continues"
50 INVERSE 0
60 PRINT "Back to normal"
```

Fig. 7.1. Special effects. INVERSE is used to introduce these effects, showing how '1' is used to switch the effect on and 'Ø' to switch off.

Take a look at the program in Fig. 7.1. The special effect in this example is INVERSE, short for 'inverse video'. It means that the colours we use for printing are reversed. Using black and white, we normally have black letters on a white background, but with INVERSE 1, we have white letters on a black background. As we'll see later, INVERSE 1 will also work if the colours are not black and white.

INVERSE is used both to switch the effect on and to switch it off, according to the number that follows INVERSE. This number must be Ø or 1 – you'll get an 'invalid colour' message if you try to use any other numbers. Line 1Ø of Fig. 7.1 prints a message, using normal display, then line 2Ø switches on the inversion. When the INVERSE instruction is placed in a line of its own, or as part of a multistatement line, it will act until it is cancelled. Because of this, lines 3Ø and 4Ø print in inverse video. Line 5Ø contains the switch-off instruction, INVERSE Ø, so that line 6Ø is printed normally.

INVERSE can also be used within a PRINT instruction, though. When this is done, the effect lasts only for as long as that line, or until it is cancelled. Figure 7.2 illustrates this point, with the inverse switched on and off in line 2Ø, on in line 3Ø, but not appearing in line 4Ø. This alternative use of INVERSE, which also applies to other

```
  10 CLS
  20 PRINT "Black on white"; INV
ERSE 1;" white on black"; INVERS
E 0;" and normal"
  30 PRINT INVERSE 1;"Inverse in
 this line"
  40 PRINT "..but not in this on
e!"
```

Fig. 7.2. Using effects within a PRINT line. The effect cannot last beyond the end of the line.

effects, allows us to use a great variety of stunning displays in our programs.

Figure 7.3 illustrates another of these 'effect' instructions, FLASH. Like INVERSE, FLASH is turned on when it is followed by a 1, and

```
10 CLS
20 PRINT "Watch this"; FLASH 1
;" flash."
30 PRINT "Flashing stops on th
is line"
40 PRINT "It can be"; FLASH 1;
" on"; FLASH 0;" or off"
```

Fig. 7.3. The FLASH effect. This is programmed in the same way as INVERSE.

off when it is followed by a 0. Also like INVERSE, the effect of FLASH depends whether it is in a line (or statement) of its own, or is part of a PRINT instruction. The flashing, like all of the other screen effects, continues for as long as the flashing characters are on the screen. In other words, once you have put the flashing (or inverse) characters on the screen, they will stay there until they scroll off or until the screen is cleared.

Write it in colour

It's time now to look at the colour instructions of Spectrum. There are three particularly important ones, which use the instruction words BORDER, PAPER and INK. We'll start with BORDER. As you might guess, this deals with the colour of the 'border', the outer part of the screen on which we can't put text. The colours that we can use are shown above the top row of keys, and we select a colour by using the number that is printed on the key. For example, red corresponds to key 2, yellow to key 6 and so on.

Figure 7.4 demonstrates the range of BORDER colours, and how BORDER is used. BORDER has to be followed by a number in the range 0 to 7. These are the numbers for the colours, as shown on the keys. We print the number at the centre of the screen each time, so

```
10 CLS : PRINT TAB 9;"border c
olours"
20 FOR n=1 TO 7
30 BORDER n: PRINT AT 11,15;n
40 PAUSE 150
50 NEXT n
```

Fig. 7.4. The range of BORDER colours.

that you can see which colour is being displayed. This is a good time to make any final tuning adjustments to your TV, because a colour picture is more fussy about tuning than a black/white one.

The next items are PAPER and INK. The names tell all, really. PAPER is used to select the colour of the background, and INK to select the colour of the letters or other shapes that you print on the paper. The range of colours is Ø to 7, as before. The numbers 8 and 9 can be used, but they don't produce colours. They are reserved for rather more advanced special effects.

Figure 7.5 illustrates PAPER and INK in use. An instruction such as PAPER 6 does not, by itself, cause colour to appear. If we print on

```
  10 PAPER 6: CLS
  20 INK 1
  30 PRINT "This is ink 1 on pa
per 6"
  40 PRINT ,,"This is "; INK 2;"
ink 2"
  50 PRINT ,, PAPER 3; INK 7;" d
ifferent paper and ink in a line
"
  60 PRINT ,, INK 9;"contrasting
"; PAPER 1;" colour"; PAPER 5;"
of ink"
```

Fig. 7.5. Using PAPER and INK instructions to provide background and foreground colours.

the screen following a PAPER 6 instruction, the background for our printing will appear in yellow, but only for the part on which we have printed. To make PAPER 6 colour a complete screen yellow, we have to follow it with a CLS instruction. That's illustrated in the first line of Fig. 7.5. Line 2Ø contains only the instruction INK 1. This causes all the following PRINT instructions to appear in this colour, which is blue. Don't expect the letters to appear in a very noticeable colour, because colour TV sets are not very good at displaying colour in small chunks. Add to that the fact that 90% of the male population is partially colour blind, and you'll see that the most impressive colour displays are the ones that use strong colours in big areas.

Back to the program, though. Line 3Ø prints in blue because of the INK 1 in line 2Ø. Line 4Ø then shows INK being used, as we used INVERSE and FLASH, in a PRINT instruction. As you might expect, this makes the instruction temporary, lasting only for the duration of the line. Line 5Ø takes this a step further. It changes both the PAPER and the INK colours in one line. Finally, line 6Ø

demonstrates the use of the 'colour' 9. This isn't a colour, just an instruction which caused the INK colour always to be a contrast to the PAPER colour. As we change the PAPER colour in the course of line 6Ø, then, the INK colour changes without any effort on our part. It's a very neat and useful dodge to ensure that the message always stands out! Note, incidentally, that no matter how you alter BORDER, PAPER and INK colours, you can always see the message at the bottom lines of the screen. That's good design!

Shaping up for games

Now let's start on graphics. In this chapter, we're dealing exclusively with low resolution graphics, which means that we can print a shape in each of the normal PRINT spaces. The shapes that we can place in this way are quite varied, but the number of positions is limited. It's the usual 32 positions per line, 22 lines per screen that we use for printing text. This can produce some quite detailed drawings, as we shall see, but the scope for moving them around is rather less.

At this point, I think it's necessary to point out that a home computer, programmed in BASIC, can't produce all of the dazzling effects that you see on arcade games machines. For one thing, the arcade machines are not programmed in BASIC, and for another, they cost many times more than a home computer. Most of them will earn enough in takings in a night to buy dozens of home computers! That doesn't mean that you can't play good games on home computers. It does mean, though, that you can forget about lightning action games, with all sorts of shapes moving fast in different directions. Horses for courses, as they say ...

Getting back to the graphics, take a look at Fig. 7.6. This shows you two ways of creating shapes that you can print on to the screen.

```
  10 BORDER 4: PAPER 1: INK 4: C
LS
  20 PRINT " "
  30 PRINT "
  40 PRINT "
  50 PRINT "
  60 PAUSE 100
  70 LET g$="
                 "
  80 PRINT AT 12,12;g$
```

Fig. 7.6. Obtaining graphics shapes from the keyboard.

Line 1Ø is straightforward, setting the border, paper and ink colours that we shall use. The novelty starts in line 2Ø. We've placed some graphics shapes between the quotes, and the important thing to look at is how we did it. After the first quote, press the CAPS SHIFT and the '9' key at the same time. This causes the cursor to appear as a flashing G, meaning that you are ready to enter graphics. The graphics shapes that you can enter now are the ones that are shown on the number keys 1 to 8. In addition, if you press CAPS SHIFT along with any one of these keys, you will get the inverse of the pattern.

The first line of graphics, then, is obtained by pressing, in turn, the 8 key, then CAPS SHIFT and 7, then 8, then 4, then 8. You then have to leave graphics mode again so that you can put in the final quote mark. You leave graphics mode by pressing CAPS SHIFT and 9 again, whereupon the flashing G will turn back to a flashing L. You can then enter the quote mark in the usual way. The next line is dealt with in the same way, using a set of the characters that are on the '6' key. There are five of these characters in line 3Ø, and another five of them in line 4Ø. Line 5Ø uses a CAPS SHIFT 5, then three 8's, then a 5.

Now try it out, ignoring line 6Ø to 8Ø for the moment. You'll see that it prints the pattern that is hinted at in the listing. This is placed at the left-hand side of the screen, because we haven't used a TAB number with our PRINT instructions. It would be a nuisance to have to do this in each line, so lines 7Ø to 8Ø demonstrate a much more useful way of printing a pattern of this type, and also of controlling its screen position. Line 6Ø simply contains a PAUSE to give you time to see what's happening, and then line 7Ø declares a string variable, g$. This string is a large one, however, because it's a set of graphics characters. There's nothing new to learn here about how to get the graphics characters, but how they are placed is new. We start with the LET g$ =" part in the usual way, then switch to graphics mode by pressing CAPS SHIFT and '9'. Put in the first five graphics shapes, just as in line 2Ø, but then go for the SPACE key. Hold this key down, and watch the cursor move across the line, to reappear one line down. Stop it when it has got to the space immediately under the first character position in the top line. This is the space that follows the first quote mark – we're aiming for the position immediately under this. Having got there, type the five chequer patterns by using the '6' key. Note that you don't have to leave graphics mode to do this. Then space along and down as before, so that you can place another row of

chequer patterns immediately under the previous set. Put in the fourth line of patterns in the same way, and then leave graphics mode, and put in the last quote.

Now what all this has done is to make a string of graphics characters and spaces. Since it's a string with the name g$, we can print it in exactly the same way as we could print any other string, as line 8Ø demonstrates. PRINT AT is a particularly useful instruction for this purpose, because it allows us to print the string at any position on the screen. It also, as we shall see shortly, allows us to animate our patterns.

Steady on, though. Before we plunge any deeper, we need to know how to design these patterns. It's not easy just to sit at the keyboard and produce patterns unless you are very artistic. For the less gifted (me especially) a bit of planning is needed. The best planning aids that you can invest in are a ruler, a 2B pencil, and a pad of graph paper. I am using a 'Guildhall' graph pad Ref. 1510 at the moment. This is scaled 1,5, and 10 mm, and it's ideal for planning Spectrum patterns. I have also used graph pads by Chartwell and other suppliers, so your stationers should be able to find something suitable for you at a reasonable price.

I use the small 5 mm × 5 mm squares on the graph paper to represent the character blocks of Spectrum. I start by sketching out the pattern that I want to use, trying to keep to the edges of the blocks, as Fig. 7.7 shows. This is a very much more elaborate shape, and I've

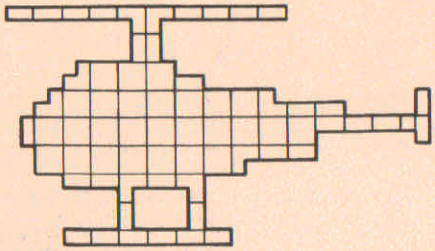


Fig. 7.7. Sketching a shape that is to be programmed. This should be done on graph paper, or on tracing paper pinned over graph paper.

used it to show just how effective these simple graphics shapes can be. Planning is important, because you will save a lot of time if you can enter the shape into the program in one go. If you make mistakes, it's very difficult to edit these graphics shapes. The reason is that the invisible spaces will be affected by deleting, and they cause the lines of

the pattern to shift about, changing the shape of the whole pattern as you edit. If you plan the shape carefully, you won't need to edit!

Figure 7.8 shows the result in the form of a program. This prints the shape at a starting position at the centre of the screen. Note that this

```
10 BORDER 0: PAPER 1: INK 6: C
LS
20 LET g$="
                                "
30 PRINT AT 12,16;g$
```

Fig. 7.8. The program which prints the shape.

does not put the shape itself at the centre of the screen. When you use PRINT AT 12,16; then what goes into that space is the first character of the shape. In this case, it's the left-hand tip of the rotor blade. If we want to print the whole shape centred, we have to find where the centre is relative to this left-hand tip. Try PRINT AT 8,10; for this shape.

Now we can try a bit of animation. The program of Fig. 7.9 shows how we can animate a shape by printing it in one position, wiping it out, and then printing in a different position. If you can print in slightly different positions each time, and repeat the action fast enough, you will see the effect of a moving image. In Fig. 7.9, the

```
10 BORDER 0: PAPER 1: INK 6: C
LS
20 LET g$="
                                "
30 FOR j=13 TO 0 STEP -1
40 PRINT AT j,10;g$
60 PRINT AT j,10; INK 1;g$
80 NEXT j
```

Fig. 7.9. Simple animation, using a string of graphics shapes which is alternately printed and wiped out.

printing is done in the usual way, and the 'wiping' is done simply by printing again with INK that is the same colour as the PAPER. The effect, however, is not ideal. It's animation of a sort, but slow and jerky. There are two reasons. One is that we have to take the shape up the screen one line at a time – we can't use half or quarter lines. The other reason is that BASIC is rather a slow-acting program language for this sort of thing. There are only a few computers that can do animation of this type convincingly using BASIC, and they cost a lot more than the Spectrum. We'll look at ways of getting around some of the problems in a moment!

Create your own characters!

The Spectrum offers another way of producing graphics, however. These are still 'low resolution' in the sense that they use the same PRINT positions on the screen, but they offer much more scope for dazzling effects. As this title suggests, we can create our own character shapes. There are three parts to this – the planning of the shapes, the instructions to the Spectrum about the shapes, and finally how we place the shapes on the screen. Let's take these three in easy stages.

We'll start, logically enough, with planning. The size of the shape we're talking about is one screen character, the size of the cursor block. Now this, and every other Spectrum character, is made out of 64 dots that are arranged on an 8 by 8 grid. Figure 7.10 shows the shape of this grid – you can re-draw it for yourself on a sheet of graph

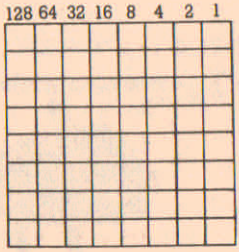


Fig. 7.10. The planning grid for designing your own characters.

paper if you want more copies. The important point is that the small squares of the grid represent positions that can be in either INK or PAPER colour, according to the value of code numbers that we use to instruct the computer.

Now the Spectrum Manual shows you one way to design these shapes and, just for the sake of choice, I'll show you another method. The key to it is the numbers that are printed on top of each column of squares. Each number is a code for the square it's over. Use the number, and the square will be in INK colour. Use 0 instead, and the square is in PAPER colour, which means invisible. An example will help to make this clearer, and it appears in Fig. 7.11. I've used a simple shape to illustrate the principle.

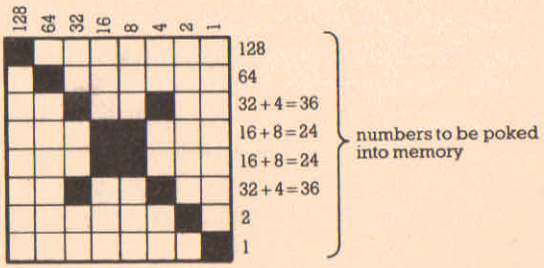


Fig. 7.11. An example of planning a shape on the grid.

The first line of squares has just one square shaded in. I usually work on tracing paper clipped over the grid pattern, but in this example, I've shown what it will look like on the graph paper itself. The shaded square in the top line is the one we want to appear in INK colour, and it's under the code number 128. There's nothing else shaded in this line, so 128 is the number we note at the side. Similarly, in the second line down, the square in the '64' position is shaded, and so that's the number we use. The third line, however, has two squares shaded. We deal with this very simply – we just add the code numbers for the shaded squares. 32 plus 4 makes 36, and that's the code number that will indicate to your Spectrum that both of these squares are to be in INK colour. The fourth line also has two squares shaded and, here again, we add the code numbers. We continue in this way until all the eight lines have been dealt with.

There are a few points to note. One is that if none of the squares in a line is shaded, the code number is zero. The other point is that you can save a lot of arithmetic by remembering that a complete set of shaded squares in a line adds up to 255. This is the maximum size of number that you can use as a code number. You must always end up with eight numbers, no matter what shape you are trying to produce.

The next matter is how we instruct the computer to produce the shape. What we have to do is to store the code numbers in the Spectrum's memory, along with a letter name that we use to obtain the shape on the screen. This makes use of two new instructions, POKE and USR. POKE has to be followed by two numbers. The first number is a reference number for a unit of memory (an address number). The second number is the code that we want to have stored there. The effect of POKE is therefore to place the code number into the memory. USR, when we use it in this way, is a method of identifying units of memory without having to search for them ourselves. We use it along with a letter to choose a part of memory that will be used when we use that letter.

Yes, an example will make it a lot easier. Figure 7.12 shows how the shape that was designed in Fig. 7.11 is put into the memory. A READ

```
  10 CLS : FOR j=0 TO 7
  20 READ k: POKE USR "a"+j,k: N
EXT j
  30 PRINT "A"
  40 PAUSE 150: CLS
  50 BORDER 0: PAPER 1: INK 7
  60 FOR j=1 TO 21
  70 PRINT AT j,22-j;"A": PAUSE
10: PRINT AT j,22-j;" ": PAUSE 1
0
  80 NEXT j
 100 DATA 128,64,36,24,24,36,2,1
```

Fig. 7.12. Storing the code numbers for a shape into the memory. The shape that is shown in line 30 was produced by going to graphics mode and then pressing the 'A' key. After this program has been run, the shape will replace the 'A'.

instruction is used to get the code numbers in order (top line to bottom line) from the DATA list. The letter we have selected to use is 'a', and by the choice of USR "a" + j, we ensure that each code is put into a different part of memory. Spectrum allows you to use the letters a to u inclusive for these special characters, so you have plenty of choice. Note that using values of j from 0 to 7 gives us eight values, equal to the number of codes we want to place in memory.

The last item is how we ensure that this will appear on the screen. Line 30 is the clue here, but all is not what it seems. The PRINT and the quote have been produced in the normal way, but the graphics shape that you see in the listing does *not* appear when you type the program. It is really a letter A! When I typed the A, I had entered graphics mode by pressing CAPS SHIFT and 9 together. What appears on the screen when you do this at first is the letter A, but what

is going to appear is our shape! What's more, after the program has been run, pressing the 'A' key in graphics mode will produce the shape, and it will also appear in the listing. That's why it's the shape you see in line 3Ø of Fig. 7.12, rather than 'A'. Like every other program listing in this book, I typed the program, tested it, printed it on the ZX printer, and recorded it.

Now we have the character, what do we do with it? Lines 4Ø to 8Ø show an attempt at animation. The technique is similar to the method that we used before, but the values that are used in the PRINT AT have been chosen to cause diagonal movement. The other point is that the 'wiping' action is done by printing a blank. If you try to print the shape again in PAPER colour, you will get an unexpected effect! This is due to the fact that INK affects the whole 64 dots of a shape.

It's still not a satisfactory animation, so let's look briefly at a dodge which can improve things for simple shapes. Figure 7.13 shows a shape which we allocate to 'a', and the same shape in two halves which are allocated to 'b' and to 'c'. When we put this on the screen (see Fig. 7.14 for the program) we start with 'a' shape. We then wipe this, and

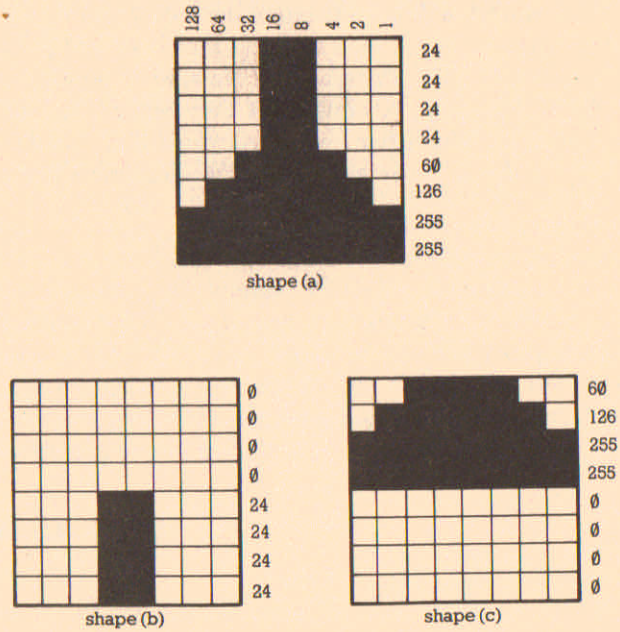


Fig. 7.13. A way of getting more convincing animation.

```
  10 CLS : FOR j=0 TO 7: READ k
  20 POKE USR "a"+j,k: NEXT j
  30 FOR j=0 TO 7: READ k: POKE
USR "b"+j,k: NEXT j
  40 FOR j=0 TO 7: READ k: POKE
USR "c"+j,k: NEXT j
  50 BORDER 0: PAPER 1: INK 4: C
LS
  60 FOR j=21 TO 1 STEP -1
  70 PRINT AT j,16;"A": PAUSE 2
  80 PRINT AT j,16;" ": PAUSE 2
  90 PRINT AT j-1,16;"B";AT j,16
;"C": PAUSE 2
 100 PRINT AT j-1,16;" ";AT j,16
;" ": PAUSE 2
 110 NEXT j
 500 DATA 24,24,24,24,60,126,255
,255
 510 DATA 0,0,0,0,24,24,24,24
 520 DATA 60,126,255,255,0,0,0,0
 600 GO TO 9999
```

Fig. 7.14. The program for Fig. 7.13.

then print the 'b' shape on the next line up. This is the nose of the rocket, but appearing on the bottom half of a character position. Below it we print 'c', which is the tail of the rocket, placed at the top half of the character position. The effect is as if we had printed the rocket shifted half a character space up! Now, when we animate this, it looks decidedly better. It would look ever better if we could use quarter shapes, but that's hard work. As you'll find, good displays always are – and that's why good games programmers can earn so much!

Chapter 8
High Resolution Graphics

High resolution graphics means picture patterns that are created with smaller units than are possible with the text size of characters. The Spectrum allows you to use high resolution graphics instructions which operate with a different screen plotting. This high resolution grid is of 256 points across the screen and 176 vertically. The picture elements in this grid are called *pixels*, and with 256 across and 176 down, that makes 256×176 =45056 pixels. Fortunately, you don't have to write programs that control each one of these pixels individually!

As always, working with high resolution graphics involves planning, and we need a planning grid. Figure 8.1 shows such a grid, with the boxes numbered in a way that is needed for Spectrum. The numbers across the grid run from ∅ to 255, and the numbers up the screen run from ∅ to 175. These are the numbers that we must use in our high resolution graphics commands. Notice that there is one very important difference between the numbering of the high resolution positions and the more familiar PRINT AT numbers. The starting point for PRINT AT is the top left-hand corner of the screen. The high resolution instructions, however, use a starting point at the bottom left-hand corner of the screen. This makes the use of high resolution graphics rather like the use of graph paper, and it also makes planning easy for anyone who has ever plotted graphs.

The Spectrum plot

PLOT is the first of the high resolution graphics instructions we have to look at. PLOT means the same as it means to anyone who is drawing a graph – put a point on the graph. PLOT has to be followed by two numbers. The first of these numbers is the 'x' number. This is

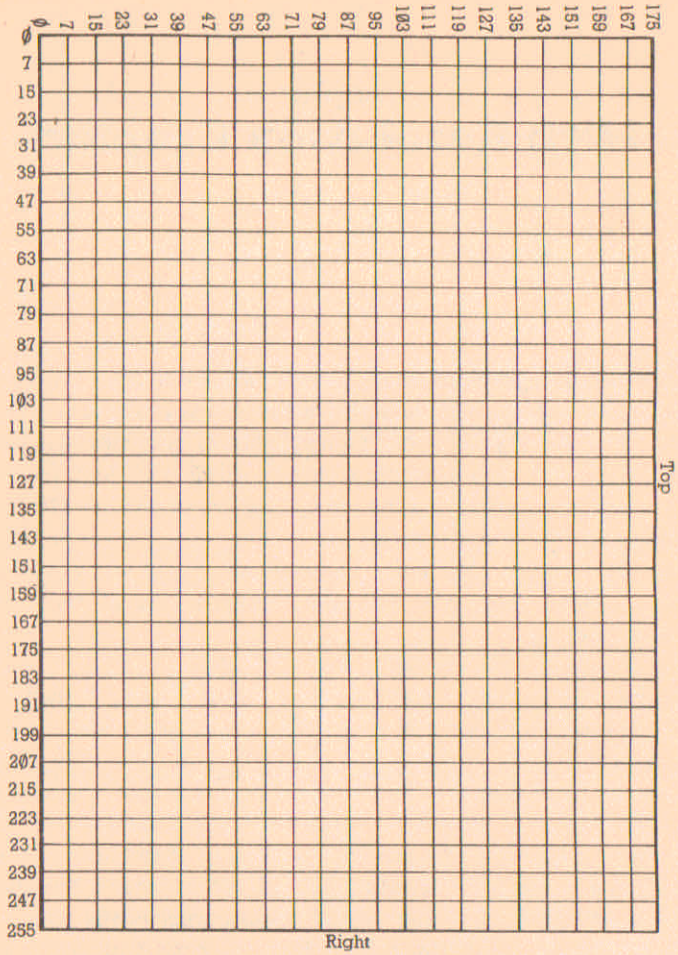


Fig. 8.1. The planning grid for high resolution graphics.

the number of pixels across the screen, starting with the left-hand side at x=Ø. The second number is the y number. This is the number of pixel places up the screen, starting with the bottom at y=Ø. The effect of PLOT 128,88 then, would be to light up a dot at the centre of the screen in whatever INK colour you happen to be using. Try it, and see just how small these pixels are compared to the character positions that we have been using.

Now we can use PLOT by itself to create some interesting effects,

but the snag about them is that they depend rather heavily on mathematics. It's rather difficult, then, to devise your own original patterns unless your mathematics is up to scratch. You can, however, try out some examples to see how PLOT operates, so we'll start with Fig. 8.2. What we are doing here is to vary a quantity 't', and plot two

```
10 CLS
20 LET v=30: LET g=10
30 FOR t=0 TO 5.6 STEP .1
40 PLOT v*t,170-g*t↑2/2
50 NEXT t
```

Fig. 8.2. Using the PLOT instruction, which prints a dot on the screen.

expressions that involve the use of 't'. The x number in the PLOT is v*t, and the y number is 17Ø-g*t↑2/2. If you have dabbled in more advanced maths or physics, you'll possibly recognise these expressions. The x one gives the distance travelled by an object that has been shot horizontally. The y expression gives the position of an object falling and accelerated by the Earth's gravity. The result of plotting these two is to show the path of something shot horizontally.

The trouble with comparatively simple expressions is that they just don't give such interesting patterns, though they are very handy if you want to plot instant graphs. To get the patterns that look really impressive, you have to use rather complicated expressions. There's another penalty to pay, too. These patterns take a long time to draw. This is because each point is having to be plotted separately, and that's a long business! Figure 8.3 shows the type of pattern that can be obtained with a lot of trial and error and a fair amount of patience. It

```
  10 BORDER 0: PAPER 1: INK 6: C
LS
  20 FOR t=0 TO 5 STEP .2
  30 FOR s=0 TO 5 STEP .2
  40 PLOT 240+SIN s*COS t,50*(CO
S s+SIN t)
  50 NEXT s: NEXT t
```

Fig. 8.3. Building up a pattern using PLOT. These patterns take a long time to create, so you might like to save them using the SAVE type of instructions illustrated at the bottom of page 144 of the Spectrum Manual.

takes a long time to build up the complete pattern, so don't lose patience, wait until the 'Ø OK, 5Ø:2' message is visible at the bottom line of the screen.

Time to draw the line

PLOT is used, apart from its applications in graph drawing, as a method of getting to a position on the screen. We talk of PLOT as 'moving the graphics cursor', meaning that it puts a dot at any part of the screen we want, and that we can then draw a pattern starting at that point. The instruction that we need to draw a line starting from a position that has been established by the PLOT instruction, is DRAW.

DRAW, followed by two numbers, will draw a straight line. The two numbers are x and y distances, respectively, but these are not used in the same way as PLOT numbers. PLOT uses what are called *absolute co-ordinates*, meaning the x and y numbers measured from the point at the bottom left-hand corner of the screen. DRAW, by contrast, uses numbers which represent the change from the previous position of the graphics cursor. For example, if we have used PLOT 128,88 to put the graphics cursor at the centre of the screen, then DRAW 10,5 will draw a line from 128,88 to 138,93 (add 10 to x, add 5 to y). If we had used DRAW −5,−10, then the line would have been drawn from 128,88 to 123,78 instead. The numbers that follow DRAW can be positive or negative, unlike the numbers that follow PLOT.

How do we set about drawing with the DRAW instruction, then? The easiest way to demonstrate this is with an example. As always, planning is the key to successful drawing, so we start with a simple geometrical pattern, drawn over the planning grid of Fig. 8.1. This star pattern is shown in Fig. 8.4(a). On this diagram, the absolute x and y numbers have been written at every place where a line changes direction. That's step 1 in planning a pattern. Step 2 is to choose a starting point. What you choose is a matter of convenience, and if you don't intend to move the pattern around, it doesn't much matter. I have picked the starting point of 130,30 for this pattern.

Once this has been done, the rest is more or less plain sailing. We can write the program lines, as Fig. 8.4(b) illustrates. Line 10 sets up the colours, and line 20 places the cursor at the chosen starting point of 130,30. The rest of the drawing consists of reading values of x and y from a DATA line, and using the DRAW instruction to create the line. Starting at 130,30, and with the next change of direction at 140,80, the difference values are 10,50. These are the values that must be placed in the DATA line, and when the line to 140,80 has been drawn, the next point at 190,90 is a change of 50,10. Eight sets of

(a)

(b)

```
  10 BORDER 0: PAPER 4: INK 1: C
LS
  20 PLOT 130,30: FOR n=1 TO 8
  30 READ x,y: DRAW x,y: NEXT n
 100 DATA 10,50,50,10,-50,10,-10
,50,-10,-50,-50,-10,50,-10,10,-5
0
```

Fig. 8.4. (a) A star pattern, showing its X and Y numbers and (b) the program for drawing the pattern.

points are needed to get back to the starting position, so the FOR... NEXT loop uses a final value of 8.

With that simple example out of the way, try something more advanced. Figure 8.5 shows a shape, along with its co-ordinate values, for which we can calculate DRAW numbers. The program which will draw this is in Fig. 8.6. Once again, I have selected a starting position – in this case, the bottom left-hand corner of the shape. The program therefore starts with PLOT 60,60 to get the cursor to this position. The next instruction, however, is RESTORE 1000. This ensures that when DATA is read, it will come from line 1000. This is not strictly necessary, since the program will go automatically to this line, but it introduces the idea that we can read data starting at any line we like. This is going to be useful!

Line 30 draws the first eighteen lines of the drawing. These take the cursor to the position 85,155, ready to draw the next tower. At this point, however, we can use a bit of cunning, because the instructions that we need to draw one tower must be the same as the instructions that we used to draw the other one. Only the starting point is different, and we're there already. If we put the DRAW numbers for

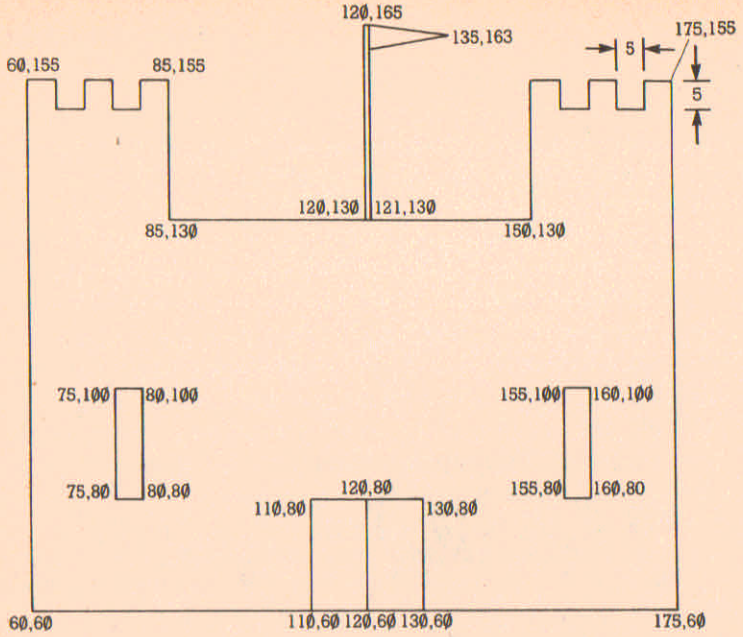


Fig. 8.5. Another shape planned on the high resolution grid, with X and Y numbers shown.

```
  10 BORDER 0: PAPER 3: INK 7: C
LS
  20 PLOT 60,60: RESTORE 1000
  30 FOR n=1 TO 18: READ x,y: DR
AW x,y
  40 NEXT n: RESTORE 1010: FOR n
=1 TO 9
  50 READ x,y: DRAW x,y: NEXT n:
 DRAW 0,-95
  60 PLOT 120,60: DRAW 0,20
  70 PLOT 120,130: DRAW 0,35: DR
AW 15,-3: DRAW -14,-3
  80 DRAW 0,-29: PLOT 75,100: GO
 SUB 500: PLOT 155,100: GO SUB 5
00
  85 GO TO 9999
  90 RESTORE 1030: FOR n=1 TO 4:
 READ x,y: DRAW x,y: NEXT n
 500 RESTORE 1030: FOR n=1 TO 4:
 READ x,y: DRAW x,y: NEXT n: RET
URN
1000 DATA 50,0,0,20,20,0,0,-20,4
5,0,0,95
1010 DATA -5,0,0,-5,-5,0,0,5,-5,
0,0,-5,-5,0,0,5,-5,0
1020 DATA 0,-25,-65,0,0,25
1030 DATA 5,0,0,-20,-5,0,0,20
```

Fig. 8.6. The program for drawing the shape of Fig. 8.5 on the screen.

the crinkly top of the tower into a separate line of DATA, we can read it again. I shall put these instructions into line 1Ø1Ø, so by using RESTORE 1Ø1Ø in line 4Ø, I can read them again. There are nine of them, so the next loop in lines 4Ø and 5Ø deals with them, and the final DRAW instruction in line 5Ø draws the left-hand wall of the castle.

We then have to put in the flagpole, flag and windows. Lines 6Ø and 7Ø deal with flagpole and flag, and we can use a bit of repetition once again for the windows. Since the windows are the same shape, we can put their DRAW numbers into one DATA line (1Ø3Ø) and use it twice. This is done by using a subroutine at line 5ØØ to draw the window shape. All that the main program has to do, in line 8Ø, is to plot the starting points and call the subroutine. The pattern is drawn impressively quickly, and it's particularly gratifying to see the detail, like the flag, appearing. Try some of these straight-line patterns for yourself!

Twinkle, twinkle ...?

The rate of drawing is not quite fast enough, however, if we want to go in for fast animation. As an example, take a look at Fig. 8.7. This

```
  10 BORDER 0: PAPER 0: INK 6: C
LS
 100 PLOT 120,50: RESTORE 1000:
GO SUB 500
 110 PLOT 120,50: RESTORE 1000:
GO SUB 600
 120 PLOT 120,80: RESTORE 1010:
GO SUB 500
 130 PLOT 120,80: RESTORE 1010:
GO SUB 600
 140 GO TO 100
 150 GO TO 9999
 500 FOR n=1 TO 8: READ x,y: DRA
W x,y: NEXT n: RETURN
 600 FOR n=1 TO 8: READ x,y: DRA
W OVER 1;x,y: NEXT n: RETURN
1000 DATA 10,40,40,10,-40,10,-10
,40,-10,-40,-40,-10,40,-10,10,-4
0
1010 DATA 5,15,15,5,-15,5,-5,15,
-5,-15,-15,-5,15,-5,5,-15
```

Fig. 8.7. Attempting animation – first try.

time, I have shown only the program, not the planning stages. The two DATA lines contain the DRAW numbers for drawing two stars – one large, the other small. The aim of the program is to draw the large star, then wipe it out, then draw the small one, wipe it in turn and

return to drawing the large star again. If this can be done quickly, it will produce a twinkle, twinkle, little star effect.

The program uses two subroutines. The subroutine in line 5ØØ is used to draw a star shape. Since both stars use eight DRAW instructions, the loop of 1 to 8 will suffice for each, and we only have to change the RESTORE instructions to cause the correct set of DATA to be read. Line 6ØØ looks identical, but contains an important difference. Between the DRAW and the x,y parts of the instruction we have OVER 1. The effect of OVER 1 is to wipe out the line. By re-drawing the lines with OVER 1 it's as if we went over the lines with an eraser. Line 1ØØ draws the large star, and line 11Ø erases it. Line 12Ø then draws the small star and line 13Ø erases it. Line 14Ø then causes the whole business to be repeated until you press the CAPS SHIFT and SPACE keys together to break the loop.

Now it twinkles, but not fast enough. You can see the star being drawn and erased again, instead of appearing and disappearing as we would like. Part of the problem is an optical illusion. Our eyes can see the anticlockwise rotation, and if we draw the stars in a different way, the effect of drawing and undrawing is not nearly so noticeable.

```
  10 BORDER 0: PAPER 0: INK 6: C
LS
  20 PLOT 120,150: DRAW 10,-40:
DRAW 40,-10: DRAW -40,-10: DRAW
-10,-40: PLOT 120,150: DRAW -10,
-40: DRAW -40,-10: DRAW 40,-10:
DRAW 10,-40
  30 CLS : GO TO 20
```

Fig. 8.8. Improving animation by a different approach (just one star this time).

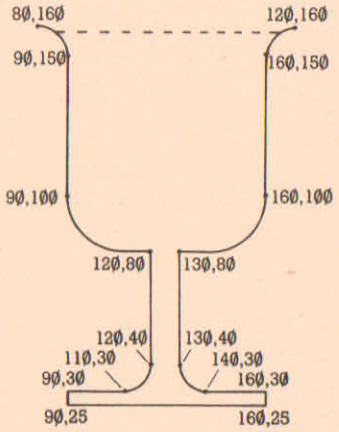
Figure 8.8 indicates how the appearance can be improved. There is just one size of star this time. It is drawn in two sections, with the direction of drawing opposite in each section. The star is erased by a CLS instruction rather than by the use of OVER 1, and we could draw a smaller star if we liked before clearing and repeating the large star.

Moving in another sector

The DRAW instruction can be followed by three numbers instead of two. When this is done, the line that is drawn is not straight, but is part of a circle! The way in which the numbers are used appears quite straightforward. The first two numbers, as before, specify the

distance between the points at the ends of the line. The third number specifies the angle through which the line turns to make a part of a circle. The first snag that you encounter when you start to use this instruction is that the angle isn't measured in degrees, but in *radians*. There are 2*pi(2π) radians to a complete circle of 360 degrees, and a right angle (90°) is pi/2 radians. As it happens, though, most of the parts of circles that we want will be semicircles (angle pi) or quarter circles (angle pi/2). What takes a lot of practice and planning is deciding what figures have to be used along with DRAW to produce the results that we need. As usual, an example tells you a lot more than instructions.

(a)



(b)

```
  10 BORDER 0: PAPER 0: INK 7: C
LS
  20 PLOT 80,160
  30 DRAW 10,-10,-PI/2
  40 DRAW 0,-50
  50 DRAW 30,-20,PI/2
  60 DRAW 0,-40
  70 DRAW -10,-10,-PI/2
  80 DRAW -20,0: DRAW 0,-5: DRAW
70,0: DRAW 0,5: DRAW -20,0
  90 DRAW -10,10,-PI/2
 100 DRAW 0,40
 110 DRAW 30,20,PI/2
 120 DRAW 0,50
 130 DRAW 10,10,-PI/2
 140 FOR x=95 TO 155 STEP 5: PLO
T INK 6;x,150: NEXT x
```

Fig. 8.9. (a) Another pattern drawn over the planning grid. This one uses partial circles. (b) The program for drawing the pattern.

Figure 8.9 shows a pattern that has been drawn, as usual, over the planning grid. The points where the lines change directions have been drawn in, so that all we need are a PLOT of the starting position, and DRAW instructions to complete the rest of the pattern. The part-circles are all obtained by using right angles (pi/2 radians), but some of them turn clockwise, and the others anticlockwise. The difference is programmed by using a negative sign for the clockwise turning, positive (or no sign) for anticlockwise. Each DRAW instruction then uses the x and y numbers that have been obtained in the usual way, by subtracting the x and y co-ordinates for the two positions, plus the pi/2 angle, positive or negative. Try the program – line 14Ø puts a liquid level in the glass!

Moving in better circles!

Using the third part of DRAW can be useful for making portions of circles, but the CIRCLE instruction is more convenient if we want to place complete circles into a drawing. CIRCLE needs three numbers following it. Of these, the first two are the x and y position numbers for the centre of the circle, and the third number is the radius. The x and y numbers are the numbers that you would use along with PLOT, and you will have to choose the radius number fairly carefully. The radius of a circle is the distance from the centre to the edge, and if you choose a number for the radius so that the line disappears off the edge of the screen, the program will stop with an error message: '5 Integer out of range'. This means that the value of radius that you have picked has been one that would place part of the circle outside the limits of the screen. To test your chosen radius, add it to the x and the y values of the starting position. If the x plus radius number is less than 256, and the y plus radius number is less than 176, all is well. Now subtract the radius number from the x and y starting numbers. If x minus radius is more than zero, all is well. If the y minus radius number is more than zero, again, all is well. If you find 'illegal' values, however, you will have to change the radius number, or shift the centre of the circle by using other x and y numbers.

```
  10 BORDER 6: PAPER 6: INK 1: C
LS
  20 FOR r=1 TO 80 STEP 4
  30 CIRCLE 128,88,r
  40 NEXT r
```

Fig. 8.10. Using the CIRCLE instruction.

Figure 8.10 illustrates circles drawn by the use of this instruction. When colours are used, you will find some odd mixing of PAPER and INK colours at parts of the circle. This is most noticeable when a set of circles are drawn using the same centre. If you omit the STEP part of the program, you will see an almost solid circle being drawn, with the colour changes rather obvious.

Being able to use circles greatly enhances our ability to produce interesting shapes, so we can now start to flex our muscles a bit.

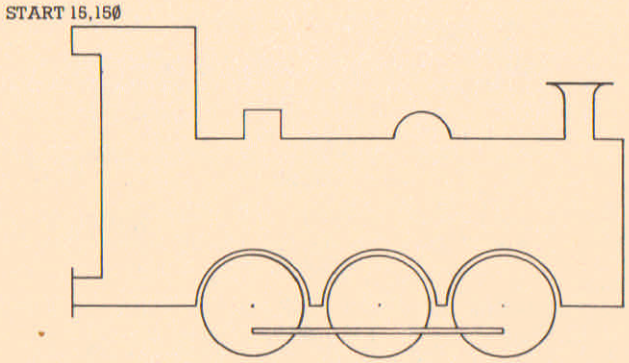


Fig. 8.11. A more elaborate shape which uses all of the drawing instructions.

Figure 8.11 shows a plan for a shape that is more elaborate than any we have tried so far. This combines the use of all of the graphics instructions, and it illustrates the CIRCLE instruction being used in a drawing. As usual, the planning stage of the program is the most important, and you should mark in the co-ordinates of all the places where lines change direction. The important point about planning a drawing of this kind is that it shows up where instructions are repeated, so that we can save a lot of programming by using loops. In this example, the wheel-arches and the wheels are shapes which repeat three times, so we should be able to use a loop to create each of these.

As usual, we pick a starting point. I have picked the top left-hand corner of the drawing, because it's the point nearest to the edge of the screen. From this point, eight straight lines are used to draw as far as the start of the first wheel arch. The DATA (Fig. 8.12) for this part of the drawing is put into line 1ØØØ. You can enter lines 1Ø, 2Ø, and part of 3Ø by themselves, along with line 1ØØØ, just to test that this part has been done according to plan. That, in fact, is what I did. It's useful to

```
  10 BORDER 6: PAPER 6: INK 1:
CLS
  20 PLOT 15,150: FOR n=1 TO 8
  30 READ x,y: DRAW x,y: NEXT n:
 FOR n=1 TO 3
  40 DRAW 40,0,-PI: DRAW 10,0: N
EXT n
  50 FOR n=1 TO 4: READ x,y: DRA
W x,y: NEXT n
  60 DRAW 10,10,-PI/2: DRAW -30,
0: DRAW 10,-10,-PI/2
  70 DRAW 0,-10: DRAW -40,0: DRA
W -30,0,PI
  80 FOR n=1 TO 7: READ x,y: DRA
W x,y: NEXT n
  90 FOR x=80 TO 180 STEP 50
 100 CIRCLE x,50,18: NEXT x
 110 PLOT 80,40: DRAW 100,0: DRA
W 0,-2: DRAW -100,0: DRAW 0,2
 120 RESTORE 1030: PLOT 200,140:
 FOR n=1 TO 3: READ x,y,r: DRAW
x,y,r: NEXT n
 130 RESTORE 1030: PLOT 200,140:
 FOR n=1 TO 3: READ x,y,r: DRAW
OVER 1;x,y,r: NEXT n
 140 GO TO 120
1000 DATA 0,-10,15,0,0,-80,-10,0
,0,5,0,-20,0,10,40,0
1010 DATA 10,0,0,60,-10,0,0,10
1020 DATA -55,0,0,5,-5,0,0,-5,-1
0,0,0,40,-45,0
1030 DATA -5,10,-PI,10,0,-PI,0,-
10,-PI
```

Fig. 8.12. The loco-drawing program.

build up a large drawing in stages like this, because you can then sort it out in stages. If you attempt too much at a time, you'll usually find it much more difficult to sort out if there has been a mistake somewhere.

The next part of the drawing is the wheel arches. Since there are three identical units, we can use a loop, and this begins in the last part of line 3Ø. The DRAW instructions are identical for each wheel arch, and these are contained in line 4Ø. The next part of straight line drawing then goes as far as the smoke-stack, just at the point where it is flared. Line 6Ø deals with the top of the smoke-stack, and line 7Ø draws as far as the dome. The rest of the main shape consists of straight lines, and it is drawn by another loop in line 8Ø.

With the outline drawn, details can then be filled in. Lines 9Ø and 1ØØ draw the wheels, using a CIRCLE instruction placed within a loop. The only value which changes from one pass of the loop to the

next is the x value of the centre of each wheel. Line 11Ø then draws the connecting rod. Just to add a flourish, lines 12Ø and 13Ø produce a wisp of smoke which is alternately drawn and erased to give the impression of our locomotive lightly steaming along.

Now for your turn! Can you add some embellishments to this? You will have to place them between lines 11Ø and 12Ø, so you might want to change the numbers of lines 12Ø, 13Ø and 14Ø (and the GOTO in line 14Ø) to make more space. The sort of thing you might want to add would be a driver's window, front buffers, and anything else you can think of. The more you can successfully add in this way, the more confidence you will have in using the high resolution instructions of your Spectrum.

Amazing activity

There are two instructions I have left until last in this chapter, POINT and ATTR. The reason is that they don't place anything on the screen. What they do is to analyse what is already on the screen, and this is a very valuable activity. If you have a high resolution graphics program, for example, that depends on moving some object across the screen, it's very handy to be able to tell when it comes up against a 'barrier'. POINT is the main instruction that we can use for this bit of diagnosis. Take a look at the program in Fig. 8.13. This starts in line

```
  10 BORDER 0: PAPER 6: INK 2: C
LS
  20 PLOT 10,10: DRAW 0,160: PLO
T 240,10: DRAW 0,160
  30 LET x=128: LET y=88: LET k=
1
  40 PLOT x,y: PAUSE 5: PLOT OVE
R 1;x,y: LET x=x+k: IF POINT (x+
k,y)=1 THEN LET k=-k
  50 GO TO 40
```

Fig. 8.13. Using POINT to detect a barrier.

2Ø by drawing two vertical lines in INK colour, one at each side of the screen. Line 3Ø then sets the position of a dot which is printed on the screen by the PLOT instruction at the start of line 4Ø. After a short pause, this point is wiped out by using OVER 1, and the value of x is changed. Since k is defined as 1 in line 3Ø, the change is from x=128 to x=129. At this place, we check to find what the next point on the screen looks like. POINT(x+k,y) will have a value of Ø if the next place along is in PAPER colour. If the next place along is in INK

colour, then the value of POINT is 1. Since the only places on the path that are in INK colour are the 'walls', then this will operate only when the dot is one space away from a wall. When this happens, the value of k is changed to −k. If, for example, x has the value of 239, and k is 1, POINT(x+k,y) will give 1, and k becomes −k. The next time we use LET x=x+k, with x=239, the value of x will become 238, not 24Ø. This is what moves the dot back across the screen when it has 'hit' one wall. At the left-hand side, the value will again reverse before the 'wall' is struck. If, incidentally, we let the value of x get to the value that it has at the wall position, the wall will show a gap when the dot moves away. This is because of the OVER 1 action, and it's useful if you want to break through walls.

The type of action that was used in Fig. 8.13, however, is more useful for bouncing objects, finding a way through a maze, and other non-destructive activities. POINT is the instruction that is always used along with high resolution graphics instructions. There is another instruction, however, in the shape of ATTR, which we use for low resolution displays. ATTR has to be followed by the two numbers, within brackets, that we would use in a PRINT AT instruction. The number that we obtain will reveal the INK and PAPER colours, and whether flashing and bright text has been commanded for that character position. Since ATTR does not work with the high resolution points, however, we won't devote further space to it here.

Chapter 9
Sound Sense

The ability to produce sound is an essential feature of all modern computers. The sound of the Spectrum comes from a very small loudspeaker, however, and is not particularly easy to hear. The version of the Spectrum available in the United States uses a sound system which gives considerably greater volume.

To start with, we can operate on the sound that we get when we press a key. This is controlled by a number contained in the memory of the Spectrum. When we switch the Spectrum on, this part of memory is set with the values that will normally be used. We can alter these values, however, by using the instruction called POKE. POKE, which we have come across before in connection with do-it-yourself character shapes, has to be followed by two numbers. The first of these numbers is the address number of the memory that we want to change. The second number is the new amount we want to store at that address. This second number must be a whole number that lies between Ø and 255, inclusive. As far as the sound of the key-click is concerned, the memory address number is 236Ø9. If you type:

```
POKE 236Ø9,1ØØ
```

and then press ENTER, you will find from then on all your key-presses are marked by a more noticeable sound. It's a great improvement on the very soft click that is the normal sound.

The wild waves

What we call sound is the result of rapid changes of the pressure of the air round our ears. We don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or *hertz*. A cycle of a wave is a set of changes, first in one direction,

then in the other and back to normal, which we can illustrate by the graphs in Fig. 9.1. The reason that we talk about a sound 'wave' is because the shape of this graph is a wave shape.

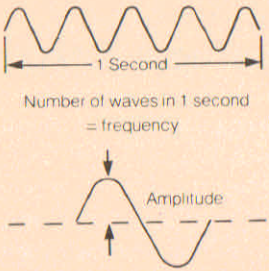


Fig. 9.1. Graphs showing a sound 'wave', illustrating amplitude and frequency.

The *frequency* of sound is its number of hertz – the number of cycles of changing air pressure per second. If this amount is less than about 20 hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15,000 hertz. The frequency of the waves corresponds to what we sense as the *pitch* of a note. A low frequency of 80 to 120 hertz corresponds to a low pitch bass note. A frequency of 400 or above corresponds to a high pitch treble note.

The amount of pressure change determines what we call the *loudness* of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

The Spectrum sound system is a simple one, and the notes from it have fixed amplitude. As I mentioned earlier, this amplitude is very low, and you may not hear the notes if there is any other sound in the same room. The electrical version of the sound signal, however, is sent out from the two cassette leads. If you connect either cassette lead to a hi-fi amplifier (or a lo-fi one, as you please!), then you can obtain the sounds of the Spectrum at any volume you like to use.

A touch of beep-bop

The Spectrum sound instruction is called BEEP, and it has to be followed by two numbers. Of these, the first number is a *duration number*. It is the number of seconds for which the note will sound. Spectrum allows the use of fractions as well as whole numbers, so this instruction does not confine you to notes of one second or longer. The second number is for the *frequency* of the sound. We don't use the actual frequency numbers, which would be rather too much to type, but numbers that provide a usable set of values.

What constitutes a useful set of values? The answer is a set based on the piano keyboard. The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes, called the 'scale of C Major'. Figure 9.2 will show you what the Spectrum makes of this scale. The scale starts on a note that is called Middle C, and ends on a note that is also called C, but which is the eighth note above middle C. A group of eight notes like this is called an *octave*, so that the note you end with in Fig. 9.2 is the C which is one octave above Middle C.

```
10 CLS : PRINT TAB 14;"BEEP"
20 FOR n=1 TO 8
30 READ k: BEEP 1,k: NEXT n
100 DATA 0,2,4,5,7,9,11,12
```

Fig. 9.2. A 'scale of C Major' program, using BEEP.

Fig. 9.3. The piano keys with the notes printed.

The appearance of these keys on the piano keyboard is illustrated in Fig. 9.3. Middle C is, logically enough, at the centre of the keyboard, and we move right for higher notes, left for lower notes. One of the complications of music, however, is that the frequencies of the notes of a scale are not evenly spaced out. The 'normal' full spacing is called a *tone* and the smaller spacing is called a *semitone*.

Spectrum allows the use of both negative and fractional numbers for its 'pitch' numbers in the BEEP instruction. This allows us to produce notes which the piano cannot play, and which can only normally be produced by instruments like the violin. More important

for us, however, is the fact that the BEEP method of controlling the sound allows us to produce some sound effects which are very useful as reminders, or in games. If you want to zap a Klingon, phaze a Martian, or make a small frog go splat on a motorway, then some sort of sound for the occasion can be generated by the BEEP instruction. It's never going to be the sort of impressive sound effects that some other computers can generate, but it's simple to work with.

Let's start with a rising pitch of note which makes a useful warning, or a 'something about to happen' note. This is illustrated in Fig. 9.4.

```
10 CLS : PRINT TAB 10;"Rising
note"
20 FOR n=-10 TO +10 STEP .2
30 BEEP .02,n: NEXT n
```

Fig. 9.4. Programming for a rising pitch note.

The loop that starts in line 2Ø uses values of n that range from −1Ø to +1Ø in steps of .2. These are the numbers that we shall use as frequency numbers in the BEEP instruction in line 3Ø. The changes will take rather a long time unless we use a fairly short time for the note, and the figure of .Ø5 has been carefully chosen. This time is critical. If you make it too long, the sound will be just a set of separate notes. If you make it too short, the notes sound too much like clicks to make an interesting sound effect. This one, like most of the sound signals that we shall be looking at in this chapter, sounds better on a hi-fi system.

```
10 CLS : PRINT TAB 13;"Warble"
20 FOR n=1 TO 50: BEEP .05,8:
BEEP .05,9: NEXT n
```

Fig. 9.5. A warbling note program.

Figure 9.5 shows a program that produces a warbling note. This is particularly useful for attracting attention, or for announcing an event in a game. For some reason, a warbling note attracts our attention more than a single note, which is why a warbling note was chosen for the later types of telephones. The warble in this program uses the loop in line 2Ø. This sounds 50 notes, which are short with a duration number of .05. The two pitch numbers that have been chosen in this example are 8 and 9. Higher pitches are even more effective, and values like 25,26 give effective attention-getting warbles.

Another way of getting attention is the sound of two notes of very different pitch. The combination of one high note and one low note sounded alternately can be a very effective way of attracting notice, and this is illustrated in Fig. 9.6. The notes that are used have the frequency numbers of 1 and 20. This is the sort of separation that is needed if the method is to sound effective. Try also a pair of warbling notes, one at 1 and 1.5 pitch numbers, the other at 25 and 26.

```
  10 CLS : PRINT TAB 13;"Hi-Lo"
  20 FOR n=1 TO 5: BEEP .05,1: B
EEP .05,20: NEXT n
```

Fig. 9.6. Sounding high and low notes alternately.

Sometimes, just a few notes of music can convey a useful message (what did we do before Colonel Bogey was composed?). Figure 9.7 shows how a short music message can be used to signify the loss of a game. There's a rich choice here, because many pieces of music are associated with moods, feelings, and plain insults! If you want to use music in your programs, however, you need to be able to read music.

```
  10 CLS : PRINT TAB 11;"Beat yo
u!"
  20 BEEP 1,7: BEEP 1,9: BEEP 1,
7: BEEP 2,10
```

Fig. 9.7. A short musical message.

```
  10 CLS : PRINT TAB 10;"Random
sound"
  20 FOR n=1 TO 50
  30 LET j=2*RND: LET p=20*RND
  40 BEEP j,p: NEXT n
```

Fig. 9.8. A 'random music' program.

Finally, take a look at Fig. 9.8. This is a program which composes music. It may not sound like music to you, but to my ear it's as good as most modern composers. Line 3Ø generates two random numbers. The number j is used as a duration number, and p is used as a note pitch number. The two of them are then used in the BEEP instruction in line 4Ø, and the results are, to say the least, interesting. It doesn't sound like Mozart, but, let's face it, who else does?

Chapter 10
The Game – Squids In

This is a game which makes use of all the techniques that have been explained in this book. The way in which the game, *SQUIDS IN*, is designed follows the advice in Chapter 6. Because the techniques should be familiar to you, only a brief outline of some points will be given here.

The first point concerns the graphics. User-defined graphics have been used for the diver, the squid, the dart and the explosion. Since this printout was taken from a working program, the shapes have been printed in lines 21Ø to 24Ø. The actual letters that have to be pressed in these lines are s in line 21Ø, m in line 22Ø, d in line 23Ø and p in line 24Ø.

The starting position of the squid is held by the values of x and y, and the starting position of the diver is held by a ,d. Both of these sets of quantities change as the two characters are moved. The wiggling action of the squid is programmed by lines 6ØØ to 65Ø, and the movement of the diver is programmed by lines 41Ø to 43Ø. The squid will kill the diver if it gets near enough, and this is ensured by line 43Ø. Similarly, the dart will kill if it gets close enough in the horizontal direction, and this is programmed in line 48Ø.

Rolling your own

You can modify this program to suit yourself. You may want to change the scoring system in lines 71Ø and 96Ø so that your scores look better! You can reduce the speed of the squid by printing it in three pieces, using the method that was explained in Chapter 7. Similarly, you can speed up the dart by using LET b=b+2 in line 46Ø. You will then have to use PRINT AT d,b−2 in line 5ØØ to ensure that the old image of the dart is wiped. If you want to make the game more

difficult, you can increase the wiggling motion of the squid; if you want to make it easier you can reduce the wiggle. This can be done by testing the value of j in line 62Ø. For example, if you program:

```
61Ø LET R=Ø
62Ø LET j=RND: IF j<.2 THEN LET R=R-1
625 IF j>.8 THEN LET R=R+1
```

this will reduce the wiggling considerably. It's your game, and you can try if you want to!

Squids In

```
  10 REM definitions
  20 LET dart=10: LET score=Ø
  30 GO SUB 1200: REM title
  40 GO SUB 1300: REM instructio
ns
  50 GO SUB 200: REM characters
  60 GO SUB 300
  70 LET pl=Ø: GO SUB 400
  80 IF dart>Ø THEN GO TO 60
  90 PRINT AT 19,8;"Your score i
s ";score
 100 PRINT "Another game? Please
 press y "/"or n"
 110 LET k$=INKEY$: IF k$="" THE
N GO TO 110
 120 IF k$="y" OR k$="Y" THEN LE
T dart=10: GO TO 60
 130 GO TO 9999
 200 FOR a=Ø TO 7: READ d,e,f,g
 210 POKE USR "a"+a,d
 220 POKE USR "b"+a,e
 230 POKE USR "c"+a,f
 240 POKE USR "d"+a,g
 250 NEXT a: RETURN
 300 PAPER 1: CLS
 310 LET x=30: LET y=12
 320 PRINT AT y,x; INK 6;CHR$ 16
2
 330 LET a=1: LET d=12: LET b=2
 340 PRINT AT d,a; INK 7;CHR$ 15
6
 350 RETURN
 400 GO SUB 600: IF x<2 THEN GO
TO 650
 410 IF INKEY$="u" THEN INK 1: P
RINT AT d,a;CHR$ 156: LET d=d-1:
 INK 7: PRINT AT d,a;CHR$ 156
 420 IF INKEY$="d" THEN INK 1: P
RINT AT d,a;CHR$ 156: LET d=d+1:
 INK 7: PRINT AT d,a;CHR$ 156
 430 IF ABS x+y-a-d<2 THEN GO SU
```

```
B 700: GO TO 540
 440 IF INKEY$<>" " THEN GO TO 4
00
 450 LET dart=dart-1
 460 LET b=b+1
 470 IF ABS (x-a)<2 AND ABS (y-d
)<2 THEN GO SUB 700: GO TO 540
 480 IF ABS (b-x)<2 AND d=y AND
b<31 THEN GO SUB 900: GO TO 540
 490 PRINT AT d,b; INK 2;CHR$ 14
7
 500 PRINT AT d,b-1; INK 1;CHR$
147
 510 GO SUB 600: IF x<2 THEN GO
TO 540
 520 IF b<31 AND x>1 THEN GO TO
460
 530 IF pl=0 THEN GO SUB 800
 540 RETURN
 600 PRINT AT y,x; INK 1;CHR$ 16
2
 610 LET R=1
 620 LET j=RND: IF j<.5 THEN LET
 R=-1
 630 LET x=x-1: LET y=y+R: IF y>
20 THEN LET y=20
 635 IF y<2 THEN LET y=1
 640 PRINT AT y,x; INK 6;CHR$ 16
2
 650 RETURN
 700 PRINT AT d,b-1; INK 1;CHR$
147: PRINT AT y,x; INK 1;CHR$ 16
2: PRINT AT d,a; INK 6;CHR$ 162:
 FOR k=1 TO 50: BEEP .02,5: BEEP
 .02,5.5: NEXT k
 710 LET score=score-20
 720 RETURN
 800 FOR k=1 TO 50: BEEP .02,5:
BEEP .02,20: NEXT k
 810 RETURN
 900 PRINT AT d,b-1; INK 1;CHR$
147: FOR k=1 TO 5: BEEP .1,20: B
EEP .1,30: NEXT k
 910 LET b=31: LET pl=1
 930 PRINT AT y,x; INK 1;CHR$ 16
2
 940 PRINT AT y,x; INK 2;CHR$ 15
9
 950 FOR k=1 TO 200: NEXT k
 960 LET score=score+100: RETURN

1000 DATA 252,16,0,129,4,16,0,66
1010 DATA 255,63,2,36,15,56,31,0
1020 DATA 15,24,2,0,255,20,0,36
1030 DATA 4,34,0,66,252,65,0,129
1200 PRINT TAB 11; FLASH 1;"SQUI
DS IN"
1210 PAUSE 150: RETURN
```

```
1300 PRINT TAB 10;"INSTRUCTIONS"
'
1310 PRINT "You are a diver arme
d with one"'"explosive dart gun.
 You are"'"being attacked by a g
iant squid"'"and your only hope
of survival"'"lies in killing it
 or avoiding"'"it. You will kill
 it if you hit"'"it in the centr
e _ use the "'"SPACE key to fire
 the dart."
1320 PRINT "  Before firing the
dart, you"'"can move up or down
using the "'"u and d keys. After
 you have"'"fired, however, you
can't move."'" The squid does no
t swim "'"straight towards you,
and it"'"may miss. If it gets ne
ar,it "'"will swallow you. You a
re allowed ten shots, and then a
 score is printed"
1330 PRINT "Press ENTER to start
": INPUT z$: RETURN
```

Chapter 11
A Data Processing Program – Datamaster

Unlike the other programs in this book, this is a very long program which was developed on a 48K Spectrum. It can be fitted into a 16K machine by omitting the instructions and by dimensioning for fewer entries, shorter entries, or both. If you are going to use your Spectrum as a serious data processing machine, however, you really need the extra memory of the 48K machine.

The program allows you to create a file of four headings. These can be anything you like, but there must be four of them. You might, for example, use headings such as NAME, PHONE NUMBER, DATE OF BIRTH, DISTANCE AWAY. Once the headings are created, you can (Menu option 2) use them to enter information under each heading. Once you have made a list of the data, you can record it. The data can be replayed, and you can list all of it, pick out one item, change, delete, or sort the list. The sorting will be in order, either of number or in alphabetical order. You can sort by any of your headings. If you choose to sort by name, for example, the names will be put into alphabetical order. If you sort by date of birth, the order will be the order of age. If you sort by distance, the order will be the order of distance, and so on.

A few precautions are needed. You should always answer 'y' to the Scroll? question. The reason is that the program will stop if you don't. This is the normal action of the Scroll? question. Another point is that if you find that the program has stopped, perhaps because you answered 'n' to Scroll?, then you can get back to the menu by typing GO TO 14Ø. If you use RUN, all of your data will be lost. Note also that when the data is recorded, the 'press any key' step occurs five times – once for the titles, and then four times for the sets of data.

The program is constructed along the lines that we dealt with in Chapter 6, so it should contain no surprises. Remember that you

don't have to type it into the machine in one go. You can type a section at a time, saving the program on tape as you go, and replaying it before you add more, until the whole program has been recorded. Since the program has been designed to be 'universal', you may want to tailor it to your own requirements. You may, for example, want to use b$ for addresses, so that you need to dimension b$ to be longer. If c$ and d$ are used for short items, like date of birth and telephone number, they could be dimensioned for shorter strings. You might not want to use all of the options, so you could delete some subroutines. You might want to be able to add values in one heading, so you would convert each item from a string back to a number by using VAL. Note that this is possible only if the string is a pure number string. You can convert '23.6' but not '23.6 miles'.

Using the database

If you have never used a database program before, an example of how this program can be used will probably be more useful to you than a description of the program. Let's imagine that you have decided to use the database to hold details of your record collection. The items that you need to know about each record are the title, the performer(s), the disk or cassette number, and the date when you bought it. if we assume that you want to use the database program as it is, then you load the program from its cassette, and RUN it.

When the menu appears, you select the first option ('Start new type of file'). This allows you to enter your titles, so you press the '1' key, without using ENTER. After a short pause, you will see the heading 'New file specification' appear, and under it the invitation to type your titles. Just below that, you will see:

```
Heading 1 is_
```

When this appears, you would type, in this example, 'Record Title'. When you press ENTER, the words:

```
Heading 2 is_
```

will appear, and you can then type 'Performer(s)' and press ENTER. You enter your other two titles in the same way.

Once this has been done, you can enter the details of all your records. The program will return to the menu after selecting titles, providing that you answer 'y' to the question: 'Do you want to return

to the menu?'. You can now press the '2' key so as to select 'Enter items in file'. You don't need to use ENTER here, because it's a one-key reply. The heading 'Enter items' will appear, with a brief reminder about how to stop entering by typing xx. The words:

Entry No. 1
Record Title

will now appear, and you can type the title of the first of your records. When you press enter, you will see:

Performer(s)

appear, and you can type the name or names. When you have entered this, you will be prompted for the disk or cassette number, which you type and enter, and finally for the date when you bought it.

When you have typed the last item for this record, and pressed ENTER, you will see:

Entry No. 2
Record Title

appear, so that you can now enter the information for the second of your records. This process will continue until you enter xx (or XX), which stops the entry process. Don't try to enter a huge number of items until you are familiar with the program. The number that you can use is limited to 50 by the dimensioning in line 1Ø, but a 16K Spectrum will not accept so many entries.

At the end of entry, the program will, as always, return to the menu when you answer 'y' to the question 'Do you want to return to the menu?'. You should then pick the recording option, because once your data is safely on tape, you won't be in any danger of losing it (by a power failure, for example) as when it is in the machine. Press the '4' key to record, and carry out the instructions. Note that you have to watch over the Spectrum as this goes on. You will be asked to 'press any key' for a total of five times to get all of the data on to a cassette. Use a fresh cassette, *not* the one you keep for the program.

Once you have done this, you still have the data in the machine, and you can see what can be done. Return to the menu, and pick item 3. This allows you to delete an item, removing all of its entries. You can also amend an item, correcting spelling or altering a number or date, without having to type all the information again. Option 7 lets you see a list of all your entries, or pick one. If you press '1' to select 'list' you

will see the first set of entries on the screen. The 'scroll?' at the bottom of the screen should *always* be answered by 'y', so that you can see the other entries in turn. If you answer 'n' to 'scroll?', the program will halt. If you do this by mistake, then press the CONT key, then ENTER (easiest method) or press the GO TO key, then 13Ø, then ENTER, which will take you back to the menu. If you do either of these things, you won't lose the data from the memory.

Do you want your records listed in alphabetical order of title? If you do, press a menu option 6. You will be asked:

Which heading do you want to use for sorting?

and you will choose 1, since you want to sort by title. After a short pause, you are asked if you want to return to the menu. The sorting has been done, and if you want to see the list in order, just take menu option 7, and '1' for list. If, on the other hand, you wanted to have your list sorted into alphabetical order of performers' names, then you would choose menu item 6, and then title 2. You could also list in order of date of buying (to show records in order of age, from oldest to youngest), or in order of record number.

Each time you use the program to work with your records data, you can now ignore menu option 1. You will simply choose option 5 each time you run the program, so that you can put in the data and work with it. You use option 1 only when you want to use the program to create another file. Perhaps you want to file your collection of banknotes, your list of British motorbikes of the fifties, the locomotives of Sir Nigel Gresley, the bridges of Thomas Telford? Whatever you want to do, the database can help you. The more you explore its uses, the more useful you will find it.

All in all, it's a program that you can play with, chop around and mould to your own needs. The current price of such programs on cassette is about £5 at the time of writing, so you have already saved yourself a bob or two!

Datamaster

```
  10 DIM a$(50,20): DIM b$(50,20
): DIM c$(50,20): DIM d$(50,20):
 DIM e$(5,20)
  20 LET m$="Please have the dat
a cassette      ready."
  30 LET n$="Watch the bottom li
ne for          instructions"
  40 LET j=1
  90 LET y$="Please answer y or
n"
 100 CLS : LET t$="DATABASE": GO
 SUB 1000: REM centre
 110 PAUSE 5: PRINT '"Do you nee
d instructions?";y$
 120 GO SUB 1050: IF k$="y" OR k
$="Y" THEN GO SUB 1100
 130 CLS : LET t$="MENU": GO SUB
 1000
 140 PRINT '"1.Start new type of
 file."'"2.Enter items in file."
'"3.Delete or change items."'"4.
Record file."'"5.Replay file."'"
6.Put in order."'"7.List or pick
."'"8.End or save program."
 150 PRINT '"Please choose by nu
mber": GO SUB 1050: LET k=VAL k$
 160 GO SUB 1800+200*k: REM sele
ct subroutine
 170 CLS : PRINT "Do you want to
 return to the"'" menu"
 180 PRINT : GO SUB 1050: IF k$=
"y" OR k$="Y" THEN GO TO 140
 200 GO TO 9999: REM end
1000 PRINT TAB 16-LEN t$/2;t$: R
ETURN
1050 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 1050
1060 RETURN
1100 CLS : LET t$="INSTRUCTIONS"
: GO SUB 1000
1105 PRINT TAB 2;"Start your fil
e by entering "'"titles. These c
ould be items "'"like NAME, ADDR
ESS and so on."'"Four headings m
ust be used, but"'"not more than
 20 letters each."
1110 PRINT ';" Once you have fix
ed the titles,"'"you can then en
ter data for"'"each heading. The
 data can be "'"recorded, and re
played, items"'"picked out, list
ed, sorted into"'"order. The sor
ting can be done"'"by heading, s
o that you can,"'"for example, s
ort alphabetically"'"by name, by
```

```
 address, in order"'"of telephon
e number, age, "'"distance, or w
hatever you have"'"used for head
ings. When the"'"list is recorde
d, the headings"'"are recorded t
oo"
1115 PRINT "Press any key_": GO
SUB 1050: RETURN
1500 INPUT q$: IF LEN q$>20 THEN
 PRINT "Too long_please change n
ow_": GO TO 1500
1505 RETURN
2000 CLS : LET t$="New file spec
ification": GO SUB 1000: PAUSE 1
00
2005 PRINT '"Now select your tit
les, using "'"ENTER. Only four t
itles can "'"be used."
2010 FOR n=1 TO 4: PRINT "Headin
g is_ ";: GO SUB 1500: PRINT q$
2025 LET e$(n)=q$
2030 NEXT n
2035 PRINT "End of specification
": PAUSE 50
2040 RETURN
2200 CLS : LET t$="Enter items":
 GO SUB 1000: PAUSE 100
2205 PRINT '"Items will be enter
ed until you"'"enter xx as the f
irst of a set."
2210 PRINT '"Entry No. ";j;'
2215 PRINT e$(1): GO SUB 1500: I
F q$="xx" OR q$="XX" THEN GO TO
2240
2220 LET a$(j)=q$: PRINT a$(j):
PRINT e$(2): GO SUB 1500: LET b$
(j)=q$: PRINT b$(j)
2225 PRINT e$(3): GO SUB 1500: L
ET c$(j)=q$: PRINT c$(j)
2230 PRINT e$(4): GO SUB 1500: L
ET d$(j)=q$: PRINT d$(j)
2235 LET j=j+1: GO TO 2210
2240 LET j=j-1: PRINT '"End of e
ntry": PAUSE 50
2245 RETURN
2400 CLS : LET t$="Changes to fi
le": GO SUB 1000
2405 PRINT '"There are ";j;" ent
ries."'"Do you want to add more?
": GO SUB 1050: IF k$="y" OR k$=
"Y" THEN GO TO 2500
2410 PRINT ': LET t$="Change or
delete": GO SUB 1000
2415 PRINT '"You need to know th
e number of"'"the item that you
want to "'"change or delete."'"P
lease press SPACE if you want"'"
```

```
a listing now.","Press any lette
r key if you","don't want a list
ing.": GO SUB 1050: IF k$=" " TH
EN GO SUB 5000
2425 CLS : LET t$="Change or del
ete": GO SUB 1000
2430 PRINT ","Please type number
 of item,","then d for delete or
  c for ","change (as 25d or 13c)
, then","press ENTER."
2435 INPUT k$: LET q$=k$(LEN k$
TO ): LET g=LEN k$-LEN q$: LET k
=VAL k$( TO g)
2440 IF k=0 THEN GO TO 2455
2445 IF q$="d" OR q$="D" THEN LE
T a$(k)="": LET b$(k)="": LET c$
(k)="": LET d$(k)="": GO TO 2505
2450 IF q$="c" OR q$="C" THEN GO
 TO 2460
2455 PRINT "Incorrect answer of
";k$;" please","try again": GO T
O 2420
2460 CLS : LET t$="Changes": GO
SUB 1000
2465 PRINT ","For each item in th
e group,","press the SPACE if y
ou don't","want to change. To ch
ange the","item,type the new ite
m , then","press ENTER."
2470 PRINT e$(1);";a$(k): GO SUB
 1500: IF q$<>CHR$ 32 THEN LET a
$(k)=q$: PRINT "Changed to- ";a$
(k)
2475 PRINT e$(2);";b$(k): GO SUB
 1500: IF q$<>CHR$ 32 THEN LET b
$(k)=q$: PRINT "Changed to- ";b$
(k)
2480 PRINT e$(3);";c$(k): GO SUB
 1500: IF q$<>CHR$ 32 THEN LET c
$(k)=q$: PRINT "Changed to- ";c$
(k)
2485 PRINT e$(4);";d$(k): GO SUB
 1500: IF q$<>CHR$ 32 THEN LET d
$(k)=q$: PRINT "Changed to- ";d$
(k)
2490 GO TO 2505
2500 LET j=j+1: GO SUB 2200
2505 RETURN
2600 CLS : LET t$="Recording Fil
e": GO SUB 1000: LET e$(5)=STR$
j
2605 PRINT m$: PRINT "Make sure
it is wound to the","correct po
sition.","n$
2610 SAVE "names" DATA e$()
2615 SAVE "datbas" DATA a$()
2620 SAVE "datbas" DATA b$()
```

```
2625 SAVE "datbas" DATA c$()
2630 SAVE "datbas" DATA d$()
2635 PRINT "Your file is now rec
orded."'"Do you want to verify i
t?"
2640 GO SUB 1050: IF k$="n" OR k
$="N" THEN GO TO 2655
2645 PRINT '"m$;" _"'"rewound. Pl
ease proceed"'"as instructed."
2650 VERIFY "names" DATA e$(): V
ERIFY "datbas" DATA a$(): VERIFY
 "datbas" DATA b$(): VERIFY "dat
bas" DATA c$(): VERIFY "datbas"
DATA d$()
2655 RETURN
2800 CLS : LET t$="Replaying fil
e": GO SUB 1000
2805 PRINT '"m$: PRINT "Make sur
e it is rewound to the"'" correc
t position, then press"'"the PLA
Y key of the recorder."';n$
2810 LOAD "names" DATA e$(): LET
 j=VAL e$(5)
2815 LOAD "datbas" DATA a$()
2820 LOAD "datbas" DATA b$()
2825 LOAD "datbas" DATA c$()
2830 LOAD "datbas" DATA d$()
2835 PRINT '"Data loaded_please
wait": PAUSE 100
2840 RETURN
3000 CLS : LET t$="Sort into ord
er": GO SUB 1000
3005 PRINT '"Which heading do yo
u want to "'"use for sorting?"
3010 PRINT "1. ";e$(1)'"2. ";e$(
2)'"3. ";e$(3);'"4. ";e$(4)
3015 PRINT "Please select by num
ber_you do"'"not need to press E
NTER."
3020 GO SUB 1050: LET k=VAL k$:
IF k<1 OR k>4 THEN PRINT "Incorr
ect choice, 1 to 4 only."'"Pleas
e try again.": GO TO 3015: GO SU
B 5500
3030 CLS : PRINT AT 5,6;"Sorting
_please wait."
3035 LET y=1
3040 LET y=2*y: IF y<j THEN GO T
O 3040
3045 LET y=INT ((y-1)/2): IF y=0
 THEN GO TO 3065
3050 LET t=j-y: FOR m=1 TO t: LE
T q=m
3055 LET z=q+y: GO SUB 6000+2*k:
 LET q=q-y: IF q>0 THEN GO TO 30
55
3060 NEXT m: GO TO 3045
```

```
3065 RETURN
3200 CLS : LET t$="List or Pick"
: GO SUB 1000
3205 PRINT '"Please select list
(l) or pick"'"(p) press key, don
't use ENTER": GO SUB 1050
3210 IF k$="l" OR k$="L" THEN GO
 SUB 5000: GO TO 3255
3215 IF k$="p" OR k$="P" THEN GO
 TO 3225
3220 PRINT "Faulty selection ple
ase try "'"again.": GO TO 3205
3225 PRINT "Please enter number
of items, "'"then press ENTER ."
3230 INPUT q
3235 PRINT e$(1);"  ";a$(q)
3240 PRINT e$(2);"  ";b$(q)
3245 PRINT e$(3);"  ";c$(q)
3250 PRINT e$(4);"  ";d$(q)
3255 PRINT '"Another one? (y or n
)": GO SUB 1050
3260 IF k$="y" OR k$="Y" THEN GO
 TO 3225
3270 RETURN
3400 CLS : LET t$="End of progra
m": GO SUB 1000
3405 PRINT '"Have you recorded y
our data?"
3410 PRINT "If you haven't, pres
s 'n' to get"'"back to the recor
d step.": GO SUB 1050: IF k$="n"
 OR k$="N" THEN GO SUB 2600
3415 PRINT '"END OF PROGRAM.": G
O TO 9999
5000 CLS : LET t$="LIST": GO SUB
 1000
5005 FOR q=1 TO j
5010 PRINT q;".  ";a$(q);" ";b$(
q);" ";c$(q);" ";d$(q)
5015 PRINT "**********************
************"
5020 NEXT q
5025 PRINT '"Want to repeat y or
 n?": GO SUB 1050: IF k$="y" OR
k$="Y" THEN GO TO 5005
5030 RETURN
6002 IF a$(z)<a$(q) THEN GO SUB
7000
6003 RETURN
6004 IF b$(z)<b$(q) THEN GO SUB
7000
6005 RETURN
6006 IF c$(z)<c$(q) THEN GO SUB
7000
6007 RETURN
6008 IF d$(z)<d$(q) THEN GO SUB
7000
```

```
6009 RETURN
7000 REM
7005 LET x$=a$(z): LET a$(z)=a$(
q): LET a$(q)=x$
7010 LET x$=b$(z): LET b$(z)=b$(
q): LET b$(q)=x$
7015 LET x$=c$(z): LET c$(z)=c$(
q): LET c$(q)=x$
7020 LET x$=d$(z): LET d$(z)=d$(
q): LET d$(q)=x$
7025 LET x$="": RETURN
```

Appendix A
Cassette Loading Problems

If you find that your own programs which you have recorded will reload perfectly, but that programs that you have bought on cassettes will not, then the fault may be in your recorder. A tape is replayed by pulling the tape past a magnetic tapehead, and this tapehead may be slightly tilted. A recording that is made on a machine whose head is not tilted will not play back satisfactorily on a machine which has a tilted head. In the same way, a recording that has been made on a machine whose head is tilted will play back on the same machine, but not on a machine whose head is correctly aligned, or one which has a head tilted in the opposite direction. The amount of tilt is adjustable, and the adjustment is fairly simple. You need only a cassette with a program that refuses to load, and a small jeweller's screwdriver.

Place the cassette in the recorder, and remove the mains lead and all the signal jack plugs. Press the PLAY key of the recorder and watch what happens. If the lid of the recorder is transparent, you will see the mechanism, including the replay tapehead, moving up to the cassette. Any adjustment will have to be made while the tapehead is in this position. Look for a small hole drilled in the casing of the recorder, just between the lower edge of the lid and the keys (Fig. A1). This hole is located exactly above the adjusting screw of the tapehead. Put the end of the screwdriver into this hole and try to locate it in the head of the adjusting screw. If you cannot grip the screw, you may have to use a different size of screwdriver. If your cassette recorder has no adjustment hole, then you will either have to take it to a specialist repairer or buy a different recorder!

Once you have found the adjusting screw and can turn it, turn up the volume control setting of the recorder, and plug the mains cable in. When you hear the sound of the tape replaying, adjust the screw so that the sound is sharp and piercing, not dull and muffled. This

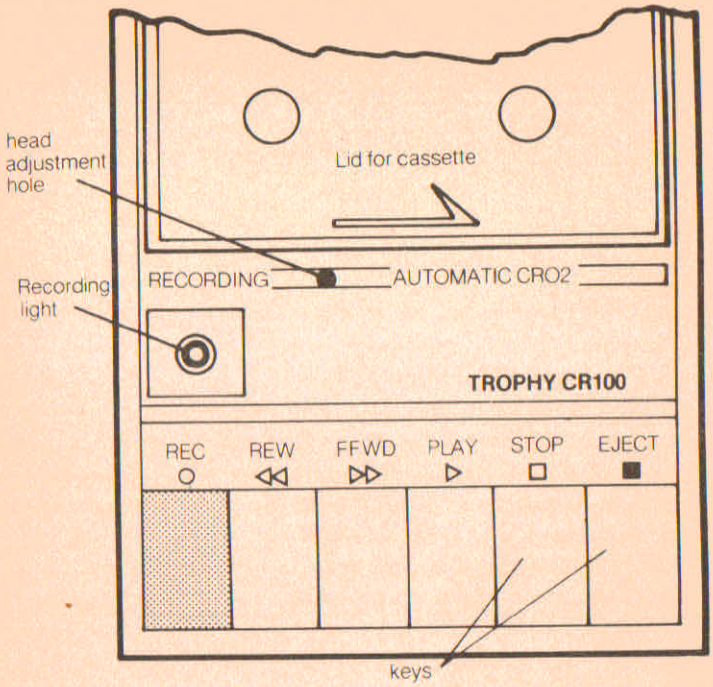


Fig. A1. The adjustment hole for the head of the CR100 cassette recorder.

should not require much adjustment, no more than one eighth of a turn of the screw either way. If adjustment makes the note seem even more muffled, then you are turning the screw in the wrong direction. You should be able to find a position in which the note is as shrill as it can be. At this position, turning the screw in either direction will make the note sound muffled. Once you have found this setting for the most shrill note, you can withdraw the screwdriver, switch off and remove the cassette. Replace the jack plugs, and try to load the troublesome program. You should find that it will load once you get the setting of the volume control correct.

The trouble now is that the cassettes which you recorded for yourself before making the adjustment may not load any longer! It doesn't always happen this way, and quite often they will load, but are more fussy about volume control settings. If you can load them, make new copies. If they will not now load, then you may have to try to find a compromise setting of the head adjustment screw which will allow you to load both your own cassettes and the bought ones. If this isn't

possible, you may have to juggle with the head adjustment, finding a setting for loading your programs, and then re-adjusting it before saving the programs again. This is such a nuisance that it's better to make any head adjustments long before you have a large stock of cassettes recorded.

Appendix B

A Useful Hint on Saving Data

Having to remove the EAR plug from the cassette recorder each time you SAVE a program or data is tedious, and is not always easy to remember. Several types of recorder will allow you to work with the EAR plug in place if a small modification is carried out to either the cassette leads or to the recorder. The modification to the lead consists of cutting the EAR plug off and replacing it with another jack plug of the same size. A miniature 330 ohm resistor is soldered across the contacts of this plug before the leads are attached. If you are experienced in electronics servicing, or know a friend who is, an alternative position for this resistor is across the switched contacts of the EAR socket of the recorder. This method has another advantage – you can hear the sound of a program loading!

The modification is simple for the Trophy and Boots recorders, and works very well though it does not appear to work on a few other makes.

Index

THE ZX SPECTRUM

The ZX Spectrum is a remarkably inexpensive and understandably popular computer with colour and sound facilities. It brings personal computing within the reach of everybody, but learning quickly how to get the best from your new machine can be difficult.

THE ZX SPECTRUM

will get the beginner off to a good start, providing many useful programs so that all the family can enjoy exploring the facilities of this versatile micro.

As well as showing you how to write programs from scratch, this book includes a complete, original, high-quality game and a practical database program designed to help you organise your home and business life.

Can you afford to be left behind in the microchip revolution? Find out about computers and programming now!

IN THE SAME SERIES FROM PANTHER BOOKS
THE ZX81
THE DRAGON 32

Front cover photograph by John Knights

HANDBOOK

U.K. £2.95 NEW ZEALAND $9.95
AUSTRALIA $9.95 (recommended)

ISBN 0-586-06104-5